

Программирование в Oracle

6-е издание

Oracle PL/SQL

ДЛЯ ПРОФЕССИОНАЛОВ

O'REILLY®
 ПИТЕР®

С. Фейерштейн
Б. Прибыл

Feuerstein Steven, Pribyl Bill

Oracle PL/SQL

Programming

6 edition

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

С. Фейерштейн, Б. Прибыл

Oracle PL/SQL

ДЛЯ ПРОФЕССИОНАЛОВ

6-е издание



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2015

ББК 32.973.233-018.2

УДК 004.65

Ф36

Фейерштейн С., Прибыл Б.

Ф36 Oracle PL/SQL. Для профессионалов. 6-е изд. — СПб.: Питер, 2015. — 1024 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-01152-5

Данная книга является подробнейшим руководством по языку PL/SQL, представляющему собой процедурное языковое расширение для SQL. В ней детально рассмотрены основы PL/SQL, структура программы, основные принципы работы с программными данными, а также методика применения операторов и инструкций для доступа к реляционным базам данных. Большое внимание уделяется вопросам безопасности, влиянию объектных технологий на PL/SQL и интеграции PL/SQL с XML и Java.

За последние 18 лет, в течение которых переиздается данная книга, она стала незаменимым руководством по PL/SQL для сотен тысяч программистов, как начинающих, так и профессионалов. Шестое издание книги полностью обновлено под версию Oracle12c.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ)

ББК 32.973.233-018.2

УДК 004.65

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-4493-2445-2 англ.

ISBN 978-5-496-01152-5

©Authorized Russian translation of the English edition of Oracle PL/SQL Programming. 6-th Edition (ISBN 9781449324452) © 2014 Steven Feuerstein Bill Pribyl. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

© Перевод на русский язык ООО Издательство «Питер», 2015

© Издание на русском языке, оформление ООО Издательство «Питер», 2015

*Моей жене Веве Сильва, ее ум, сила, красота
и искусство обогатили мою жизнь.*

— Стивен Фейерштейн

*Моей жене Норме. Она заставляет мое сердце
биться чаще уже четверть века.*

— Билл Прибыл

Краткое оглавление

ЧАСТЬ I. ПРОГРАММИРОВАНИЕ НА PL/SQL

| | |
|---|----|
| Глава 1. Введение в PL/SQL..... | 36 |
| Глава 2. Написание и запуск кода PL/SQL | 50 |
| Глава 3. Основы языка..... | 75 |

ЧАСТЬ II. СТРУКТУРА ПРОГРАММЫ PL/SQL

| | |
|---|-----|
| Глава 4. Условные команды и переходы..... | 98 |
| Глава 5. Циклы | 114 |
| Глава 6. Обработка исключений..... | 132 |

ЧАСТЬ III. РАБОТА С ДАННЫМИ В PL/SQL

| | |
|---|-----|
| Глава 7. Работа с данными в программах..... | 166 |
| Глава 8. Строки..... | 186 |
| Глава 9. Числа | 218 |
| Глава 10. Дата и время..... | 246 |
| Глава 11. Записи..... | 281 |
| Глава 12. Коллекции..... | 295 |
| Глава 13. Другие типы данных..... | 353 |

ЧАСТЬ IV. SQL И PL/SQL

| | |
|--|-----|
| Глава 14. DML и управление транзакциями..... | 386 |
| Глава 15. Выборка данных..... | 404 |
| Глава 16. Динамический SQL и динамический PL/SQL | 444 |

ЧАСТЬ V. СОЗДАНИЕ ПРИЛОЖЕНИЙ PL/SQL

| | |
|--|-----|
| Глава 17. Процедуры, функции и параметры | 486 |
| Глава 18. Пакеты | 531 |
| Глава 19. Триггеры | 559 |
| Глава 20. Управление приложениями PL/SQL..... | 606 |
| Глава 21. Оптимизация приложений PL/SQL | 664 |
| Глава 22. Ввод/вывод в PL/SQL..... | 742 |

ЧАСТЬ VI. ОСОБЫЕ ВОЗМОЖНОСТИ PL/SQL

| | |
|---------------------------------------|-----|
| Глава 23. Безопасность и PL/SQL | 778 |
| Глава 24. Архитектура PL/SQL..... | 831 |

| | |
|--|------|
| Глава 25. Глобализация и локализация в PL/SQL | 876 |
| Глава 26. Объектно-ориентированные возможности PL/SQL..... | 910 |
| Глава 27. Вызов Java-программ из PL/SQL..... | 958 |
| Глава 28. Внешние процедуры..... | 987 |
| Приложение А. Параметры функций и метасимволы регулярных выражений | 1012 |
| Приложение Б. Числовые форматы | 1017 |
| Приложение В. Маска формата даты | 1020 |

Оглавление

| | |
|-------------------------------------|-----------|
| Предисловие | 26 |
| Основные цели | 27 |
| Структура книги | 27 |
| О содержании | 28 |
| Какие темы не рассматриваются | 30 |
| Платформа и версия | 30 |
| О программном коде | 30 |
| Использование примеров кода | 31 |
| Благодарности | 31 |
| От издательства | 34 |

ЧАСТЬ I. ПРОГРАММИРОВАНИЕ НА PL/SQL

| | |
|---|-----------|
| Глава 1. Введение в PL/SQL | 36 |
| Что такое PL/SQL? | 36 |
| История PL/SQL | 37 |
| Истоки PL/SQL | 37 |
| Улучшение переносимости приложений | 38 |
| Улучшенная защита приложений и защита целостности транзакций | 38 |
| Скромное начало, постоянное усовершенствование | 38 |
| Итак, PL/SQL | 39 |
| Интеграция с SQL | 39 |
| Управляющие конструкции и логические условия | 40 |
| Обработка ошибок | 41 |
| О версиях | 42 |
| Новые возможности PL/SQL в Oracle Database 12c | 44 |
| Ресурсы для разработчиков PL/SQL | 45 |
| Книги о PL/SQL от O'Reilly | 46 |
| PL/SQL в Интернете | 46 |
| Несколько советов | 47 |
| Не торопитесь! | 47 |
| Не бойтесь обращаться за помощью | 48 |
| Поощряйте творческий (и даже радикальный) подход к разработке | 49 |
| Глава 2. Написание и запуск кода PL/SQL | 50 |
| Перемещение по базе данных | 50 |
| Создание и редактирование исходного кода | 51 |
| SQL*Plus | 51 |
| Запуск SQL*Plus | 53 |
| Выполнение SQL-инструкции | 54 |
| Запуск программы на языке PL/SQL | 54 |
| Запуск сценария | 56 |
| Что такое «текущий каталог»? | 57 |
| Другие задачи SQL*Plus | 58 |

| | |
|--|-----------|
| Обработка ошибок в SQL*Plus | 61 |
| Достоинства и недостатки SQL*Plus | 61 |
| Базовые операции PL/SQL | 62 |
| Создание хранимой программы | 62 |
| Выполнение хранимой программы | 65 |
| Вывод хранимых программ | 66 |
| Управление привилегиями и создание синонимов хранимых программ | 66 |
| Удаление хранимой программы | 67 |
| Скрытие исходного кода хранимой программы | 68 |
| Средства разработки для PL/SQL | 68 |
| Вызов кода PL/SQL из других языков | 69 |
| С, с использованием прекомпилятора Oracle (Pro*C) | 70 |
| Java, с использованием JDBC | 71 |
| Perl, с использованием Perl DBI и DBD::Oracle | 71 |
| PHP, с использованием расширений Oracle | 72 |
| PL/SQL Server Pages | 73 |
| Что же дальше? | 74 |
| Глава 3. Основы языка | 75 |
| Структура блока PL/SQL | 75 |
| Анонимные блоки | 76 |
| Именованные блоки | 78 |
| Вложенные блоки | 78 |
| Область действия | 79 |
| Уточнение ссылок на переменные и столбцы в командах SQL | 80 |
| Видимость | 82 |
| Набор символов PL/SQL | 84 |
| Идентификатор | 85 |
| Ключевые слова | 87 |
| Пропуски и ключевые слова | 88 |
| Литералы | 88 |
| NULL | 89 |
| Одинарные кавычки внутри строки | 90 |
| Числовые литералы | 91 |
| Логические (булевские) литералы | 91 |
| Точка с запятой как разделитель | 92 |
| Комментарии | 93 |
| Однострочные комментарии | 93 |
| Многострочные комментарии | 93 |
| Ключевое слово PRAGMA | 94 |
| Метки | 94 |

ЧАСТЬ II. СТРУКТУРА ПРОГРАММЫ PL/SQL

| | |
|---|-----------|
| Глава 4. Условные команды и переходы | 98 |
| Команды IF | 98 |
| Комбинация IF-THEN | 98 |
| Конструкция IF-THEN-ELSE | 100 |

| | |
|--|------------|
| Конструкция IF-THEN-ELSIF | 101 |
| Ловушки синтаксиса IF | 102 |
| Вложенные команды IF | 103 |
| Ускоренное вычисление | 104 |
| Команды и выражения CASE | 105 |
| Простые команды CASE | 106 |
| Поисковая команда CASE | 107 |
| Вложенные команды CASE | 109 |
| Выражения CASE | 110 |
| Команда GOTO | 111 |
| Команда NULL | 112 |
| Удобочитаемость кода | 112 |
| Использование NULL после метки | 113 |
| Глава 5. Циклы | 114 |
| Основы циклов | 114 |
| Примеры разных циклов | 114 |
| Структура циклов PL/SQL | 116 |
| Простой цикл | 116 |
| Завершение простого цикла: EXIT и EXIT WHEN | 117 |
| Моделирование цикла REPEAT UNTIL | 118 |
| Бесконечный цикл | 118 |
| Цикл WHILE | 118 |
| Цикл FOR со счетчиком | 120 |
| Правила для циклов FOR с числовым счетчиком | 121 |
| Примеры циклов FOR с числовым счетчиком | 121 |
| Нетривиальные приращения | 122 |
| Цикл FOR с курсором | 123 |
| Примеры цикла FOR с курсором | 124 |
| Метки циклов | 125 |
| Команда CONTINUE | 125 |
| Полезные советы | 128 |
| Используйте понятные имена для счетчиков циклов | 128 |
| Корректно выходите из цикла | 129 |
| Получение информации о выполнении цикла FOR | 130 |
| Команда SQL как цикл | 130 |
| Глава 6. Обработка исключений | 132 |
| Основные концепции и терминология обработки исключений | 132 |
| Определение исключений | 134 |
| Объявление именованных исключений | 134 |
| Связывание имени исключения с кодом ошибки | 135 |
| Именованные системные исключения | 138 |
| Область действия исключения | 139 |
| Инициирование исключений | 140 |
| Команда RAISE | 140 |
| Процедура RAISE_APPLICATION_ERROR | 141 |

| | |
|--|-----|
| Обработка исключений..... | 142 |
| Встроенные функции ошибок | 143 |
| Объединение нескольких исключений в одном обработчике..... | 147 |
| Необработанные исключения..... | 148 |
| Передача необработанного исключения | 148 |
| Продолжение выполнения после исключений..... | 150 |
| Написание раздела WHEN OTHERS..... | 152 |
| Построение эффективной архитектуры управления ошибками..... | 154 |
| Определение стратегии управления ошибками..... | 154 |
| Стандартизация обработки разных типов исключений..... | 155 |
| Коды ошибок, связанные с конкретным приложением..... | 158 |
| Стандартизация обработки ошибок | 158 |
| Работа с «объектами» исключений..... | 160 |
| Создание стандартного шаблона для обобщенной обработки ошибок..... | 162 |
| Оптимальная организация обработки ошибок в PL/SQL..... | 163 |

ЧАСТЬ III. РАБОТА С ДАННЫМИ В PL/SQL

Глава 7. Работа с данными в программах 166

| | |
|--|-----|
| Присваивание имен | 166 |
| Обзор типов данных PL/SQL..... | 168 |
| Символьные типы данных | 168 |
| Числовые типы данных..... | 169 |
| Дата, время и интервалы | 170 |
| Логические данные..... | 170 |
| Двоичные данные..... | 170 |
| Типы данных ROWID и UROWID | 171 |
| Тип данных REF CURSOR..... | 171 |
| Типы данных для поддержки интернет-технологий | 171 |
| Типы данных «Апу» | 172 |
| Пользовательские типы данных | 172 |
| Объявление данных в программе | 172 |
| Объявление переменной..... | 172 |
| Объявление константы | 173 |
| NOT NULL | 174 |
| Объявления с привязкой..... | 174 |
| Привязка к курсорам и таблицам..... | 176 |
| Преимущества объявлений с привязкой | 176 |
| Синхронизация со столбцами таблицы базы данных | 177 |
| Объявления с привязкой и ограничение NOT NULL..... | 178 |
| Подтипы данных, определяемые программистом | 178 |
| Преобразования типов данных..... | 179 |
| Неявное преобразование типов | 179 |
| Явное преобразование типов | 181 |

Глава 8. Строки 186

| | |
|----------------------------|-----|
| Строковые типы данных..... | 186 |
| Тип данных VARCHAR2..... | 187 |
| Тип данных CHAR..... | 187 |

| | |
|---|------------|
| Строковые подтипы | 188 |
| О работе со строками | 189 |
| Определение строковых констант | 189 |
| Непечатаемые символы | 190 |
| Конкатенация строк | 192 |
| Преобразование регистра | 192 |
| Традиционный поиск и замена | 195 |
| Дополнение | 197 |
| Усечение строк | 198 |
| Поиск и замена с использованием регулярных выражений | 199 |
| Работа с пустыми строками | 208 |
| Смешение значений CHAR и VARCHAR2 | 209 |
| Краткая сводка строковых функций | 211 |
| Глава 9. Числа | 218 |
| Числовые типы данных | 218 |
| Тип NUMBER | 219 |
| Тип PLS_INTEGER | 223 |
| Тип BINARY_INTEGER | 224 |
| Тип SIMPLE_INTEGER | 224 |
| Типы BINARY_FLOAT и BINARY_DOUBLE | 225 |
| Типы SIMPLE_FLOAT и SIMPLE_DOUBLE | 229 |
| Числовые подтипы | 229 |
| Числовые преобразования | 230 |
| Функция TO_NUMBER | 231 |
| Функция TO_CHAR | 233 |
| Округление при преобразовании чисел в символьные строки | 236 |
| Функция CAST | 238 |
| Неявные преобразования | 238 |
| Числовые операторы | 240 |
| Числовые функции | 241 |
| Функции округления и усечения | 241 |
| Тригонометрические функции | 241 |
| Сводка числовых функций | 242 |
| Глава 10. Дата и время | 246 |
| Типы данных даты и времени | 246 |
| Объявление переменных даты и времени | 248 |
| Выбор типа данных | 249 |
| Получение текущей даты и времени | 250 |
| Типы данных INTERVAL | 252 |
| Объявление интервальных переменных | 253 |
| Когда используются типы INTERVAL | 253 |
| Преобразование даты и времени | 255 |
| Преобразование строк в даты | 255 |
| Преобразование даты в строку | 257 |
| Часовые пояса | 259 |
| Точное совпадение маски форматирования | 262 |
| Ослабление требований к точности совпадения | 262 |

| | |
|---|------------|
| Интерпретация года из двух цифр | 263 |
| Преобразование часовых поясов в символьные строки | 264 |
| Дополнение вывода с модификатором FM | 265 |
| Литералы типа DATE и TIMESTAMP | 265 |
| Преобразования интервалов | 266 |
| Преобразование чисел в интервалы | 267 |
| Преобразование строк в интервалы | 267 |
| Форматирование интервалов для вывода | 268 |
| Литералы типа INTERVAL | 269 |
| CAST и EXTRACT | 270 |
| Функция CAST | 270 |
| Функция EXTRACT | 271 |
| Арифметические операции над значениями даты/времени | 272 |
| Операции с типами TIMESTAMP и INTERVAL | 272 |
| Операции с типом DATE | 273 |
| Вычисление интервала между двумя значениями DATE | 274 |
| Смешанное использование DATE и TIMESTAMP | 275 |
| Сложение и вычитание интервалов | 276 |
| Умножение и деление интервалов | 277 |
| Типы данных INTERVAL без ограничений | 277 |
| Функции для работы с датой/временем | 278 |
| Глава 11. Записи | 281 |
| Записи в PL/SQL | 281 |
| Преимущества использования записей | 282 |
| Объявление записей | 283 |
| Записи, определяемые программистом | 284 |
| Обработка записей | 287 |
| Сравнение записей | 292 |
| Триггерные псевдозаписи | 293 |
| Глава 12. Коллекции | 295 |
| Знакомство с коллекциями | 296 |
| Концепции и терминология | 296 |
| Разновидности коллекций | 298 |
| Примеры коллекций | 298 |
| Использование коллекций | 301 |
| Выбор типа коллекции | 305 |
| Встроенные методы коллекций | 307 |
| Метод COUNT | 308 |
| Метод DELETE | 308 |
| Метод EXISTS | 310 |
| Метод EXTEND | 310 |
| Методы FIRST и LAST | 311 |
| Метод LIMIT | 312 |
| Методы PRIOR и NEXT | 312 |
| Метод TRIM | 313 |
| Работа с коллекциями | 314 |
| Объявление типов коллекций | 314 |

| | |
|--|------------|
| Заполнение коллекций данными | 321 |
| Обращение к данным в коллекциях | 325 |
| Коллекции со строковыми индексами | 326 |
| Коллекции составных типов данных | 330 |
| Многоуровневые коллекции | 333 |
| Работа с коллекциями в SQL | 340 |
| Операции мультимножеств с вложенными таблицами | 346 |
| Проверка равенства и принадлежности вложенных таблиц | 348 |
| Проверка принадлежности элемента вложенной таблице | 349 |
| Высокоуровневые операции с множествами | 349 |
| Обработка дубликатов во вложенной таблице | 350 |
| Управление коллекциями уровня схемы | 351 |
| Необходимые привилегии | 352 |
| Коллекции и словарь данных | 352 |
| Глава 13. Другие типы данных | 353 |
| Тип данных BOOLEAN | 353 |
| Тип данных RAW | 354 |
| Типы данных UROWID и ROWID | 355 |
| Получение идентификаторов строк | 356 |
| Использование идентификаторов строк | 356 |
| Большие объекты данных | 357 |
| Работа с большими объектами | 358 |
| Понятие локатора LOB | 359 |
| Большие объекты — пустые и равные NULL | 361 |
| Запись данных в объекты LOB | 362 |
| Чтение данных из объектов LOB | 364 |
| Особенности типа BFILE | 365 |
| SecureFiles и BasicFiles | 369 |
| Временные объекты LOB | 371 |
| Встроенные операции LOB | 374 |
| Функции преобразования объектов LOB | 377 |
| Предопределенные объектные типы | 378 |
| Тип XMLType | 378 |
| Типы данных URI | 381 |
| Типы данных Any | 382 |

ЧАСТЬ IV. SQL И PL/SQL

| | |
|--|------------|
| Глава 14. DML и управление транзакциями | 386 |
| DML в PL/SQL | 387 |
| Краткое введение в DML | 387 |
| Секция RETURNING в командах DML | 391 |
| DML и обработка исключений | 391 |
| DML и записи | 392 |
| Управление транзакциями | 395 |
| Команда COMMIT | 395 |
| Команда ROLLBACK | 395 |
| Команда SAVEPOINT | 396 |

| | |
|---|------------|
| Команда SET TRANSACTION..... | 396 |
| Команда LOCK TABLE..... | 397 |
| Автономные транзакции..... | 398 |
| Определение автономной транзакции..... | 398 |
| Правила и ограничения на использование автономных транзакций..... | 399 |
| Область видимости транзакций..... | 400 |
| В каких случаях следует применять автономные транзакции..... | 400 |
| Создание механизма автономного протоколирования..... | 401 |
| Глава 15. Выборка данных | 404 |
| Основные принципы работы с курсорами | 405 |
| Терминология..... | 405 |
| Типичные операции с запросами и курсорами..... | 406 |
| Знакомство с атрибутами курсоров..... | 407 |
| Ссылки на переменные PL/SQL в курсорах..... | 409 |
| Выбор между явным и неявным курсорами..... | 410 |
| Работа с неявными курсорами..... | 410 |
| Примеры неявных курсоров..... | 411 |
| Обработка ошибок при использовании неявных курсоров | 412 |
| Атрибуты неявных курсоров..... | 414 |
| Работа с явными курсорами..... | 415 |
| Объявление явного курсора..... | 416 |
| Открытие явного курсора..... | 418 |
| Выборка данных из явного курсора..... | 419 |
| Псевдонимы столбцов явного курсора..... | 420 |
| Закрытие явного курсора..... | 421 |
| Атрибуты явных курсоров..... | 422 |
| Параметры курсора | 424 |
| Команда SELECT...FOR UPDATE..... | 426 |
| Снятие блокировок командой COMMIT..... | 427 |
| Предложение WHERE CURRENT OF..... | 428 |
| Курсорные переменные и REF CURSOR..... | 429 |
| Когда используются курсорные переменные?..... | 430 |
| Сходство со статическими курсорами..... | 430 |
| Объявление типов REF CURSOR..... | 431 |
| Объявление курсорной переменной | 432 |
| Открытие курсорной переменной..... | 433 |
| Выборка данных из курсорной переменной..... | 434 |
| Правила использования курсорных переменных..... | 436 |
| Передача курсорных переменных в аргументах..... | 438 |
| Ограничения на использование курсорных переменных | 440 |
| Курсорные выражения | 440 |
| Использование курсорных выражений..... | 441 |
| Ограничения, связанные с курсорными выражениями | 442 |
| Глава 16. Динамический SQL и динамический PL/SQL..... | 444 |
| Команды NDS..... | 445 |
| Команда EXECUTE IMMEDIATE..... | 445 |

| | |
|--|-----|
| Команда OPEN FOR..... | 448 |
| О четырех категориях динамического SQL..... | 452 |
| Передача параметров..... | 454 |
| Режимы передачи параметров..... | 455 |
| Дублирование формальных параметров..... | 456 |
| Передача значений NULL..... | 457 |
| Работа с объектами и коллекциями..... | 457 |
| Динамический PL/SQL..... | 459 |
| Построение динамических блоков PL/SQL..... | 460 |
| Замена повторяющегося кода динамическими блоками..... | 461 |
| Рекомендации для NDS..... | 462 |
| Используйте права вызывающего для совместно используемых программ..... | 462 |
| Прогнозирование и обработка динамических ошибок..... | 463 |
| Параметры вместо конкатенации..... | 465 |
| Минимизация опасности внедрения кода..... | 466 |
| Когда следует использовать DBMS_SQL..... | 469 |
| Получение информации о столбцах запроса..... | 469 |
| Поддержка требований категории 4..... | 470 |
| Минимальный разбор динамических курсоров..... | 476 |
| Новые возможности Oracle11g..... | 477 |
| Усовершенствованная модель безопасности DBMS_SQL..... | 480 |

ЧАСТЬ V. СОЗДАНИЕ ПРИЛОЖЕНИЙ PL/SQL

| | |
|--|------------|
| Глава 17. Процедуры, функции и параметры | 486 |
| Модульный код..... | 486 |
| Процедуры..... | 488 |
| Вызов процедуры..... | 489 |
| Заголовок процедуры..... | 489 |
| Тело процедуры..... | 490 |
| Метка END..... | 490 |
| Команда RETURN..... | 490 |
| Функции..... | 490 |
| Структура функции..... | 491 |
| Возвращаемый тип..... | 493 |
| Метка END..... | 493 |
| Вызов функции..... | 494 |
| Функции без параметров..... | 495 |
| Заголовок функции..... | 495 |
| Тело функции..... | 495 |
| Команда RETURN..... | 496 |
| Параметры..... | 497 |
| Определение параметров..... | 498 |
| Формальные и фактические параметры..... | 498 |
| Режимы передачи параметров..... | 499 |
| Связывание формальных и фактических параметров в PL/SQL..... | 502 |
| Значения по умолчанию..... | 505 |

| | |
|--|------------|
| Локальные модули..... | 506 |
| Преимущества локальных модулей..... | 506 |
| Область действия локальных модулей..... | 509 |
| Вложенные подпрограммы..... | 509 |
| Перегрузка подпрограмм..... | 510 |
| Преимущества перегрузки..... | 511 |
| Ограничения на использование перегрузки..... | 513 |
| Перегрузка числовых типов..... | 514 |
| Опережающие объявления..... | 514 |
| Дополнительные вопросы..... | 515 |
| Вызов пользовательских функций в SQL..... | 515 |
| Табличные функции..... | 520 |
| Детерминированные функции..... | 528 |
| Результаты неявных курсоров (Oracle Database 12c)..... | 529 |
| Модульный подход — в жизнь!..... | 530 |
| Глава 18. Пакеты | 531 |
| Для чего нужны пакеты?..... | 531 |
| Демонстрация возможностей пакетов..... | 532 |
| Основные концепции пакетов..... | 534 |
| Графическое представление приватности..... | 535 |
| Правила построения пакетов..... | 536 |
| Спецификация пакета..... | 536 |
| Тело пакета..... | 538 |
| Инициализация пакетов..... | 539 |
| Правила вызова элементов пакета..... | 543 |
| Работа с данными пакета..... | 544 |
| Глобальные данные в сеансе Oracle..... | 544 |
| Глобальные общедоступные данные..... | 544 |
| Пакетные курсоры..... | 545 |
| Повторно инициализируемые пакеты..... | 549 |
| Когда используются пакеты..... | 551 |
| Инкапсуляция доступа к данным..... | 551 |
| Исключение жесткого кодирования литералов..... | 554 |
| Устранение недостатков встроенных функций..... | 555 |
| Группировка логически связанных функций..... | 556 |
| Кэширование статических данных сеанса для ускорения работы приложения..... | 557 |
| Пакеты и объектные типы..... | 557 |
| Глава 19. Триггеры | 559 |
| Триггеры уровня команд DML..... | 560 |
| Основные концепции триггеров..... | 560 |
| Создание триггера DML..... | 562 |
| Пример триггера DML..... | 566 |
| Однотипные триггеры..... | 571 |
| Очередность вызова триггеров..... | 572 |
| Ошибки при изменении таблицы..... | 573 |
| Составные триггеры..... | 574 |

| | |
|---|------------|
| Триггеры уровня DDL | 577 |
| Создание триггера DDL | 577 |
| События триггеров | 579 |
| Атрибутные функции | 580 |
| Применение событий и атрибутов | 581 |
| Можно ли удалить неудаляемое? | 584 |
| Триггер INSTEAD OF CREATE | 584 |
| Триггеры событий базы данных | 585 |
| Создание триггера события базы данных | 586 |
| Триггер STARTUP | 586 |
| Триггеры SHUTDOWN | 587 |
| Триггер LOGON | 587 |
| Триггеры LOGOFF | 587 |
| Триггеры SERVERERROR | 588 |
| Триггеры INSTEAD OF | 591 |
| Создание триггера INSTEAD OF | 591 |
| Триггер INSTEAD OF INSERT | 592 |
| Триггер INSTEAD OF UPDATE | 594 |
| Триггер INSTEAD OF DELETE | 595 |
| Заполнение таблиц | 595 |
| Триггеры INSTEAD OF для вложенных таблиц | 595 |
| Триггеры AFTER SUSPEND | 597 |
| Настройка для триггера AFTER SUSPEND | 597 |
| Код триггера | 599 |
| Функция ORA_SPACE_ERROR_INFO | 600 |
| Пакет DBMS_RESUMABLE | 601 |
| Многократное срабатывание | 602 |
| Исправлять или не исправлять? | 602 |
| Сопровождение триггеров | 603 |
| Отключение, включение и удаление триггеров | 603 |
| Создание отключенных триггеров | 604 |
| Просмотр триггеров | 604 |
| Проверка работоспособности триггера | 605 |
| Глава 20. Управление приложениями PL/SQL | 606 |
| Управление программным кодом в базе данных | 607 |
| Представления словаря данных | 607 |
| Вывод информации о хранимых объектах | 608 |
| Вывод и поиск исходного кода | 609 |
| Проверка ограничений размера | 610 |
| Получение свойств хранимого кода | 610 |
| Анализ и изменение состояний триггеров | 611 |
| Анализ аргументов | 611 |
| Анализ использования идентификаторов (Oracle Database 11g) | 613 |
| Управление зависимостями и перекомпиляция | 615 |
| Анализ зависимостей с использованием представлений словаря данных | 616 |
| Детализация зависимостей (Oracle11g) | 619 |

| | |
|---|------------|
| Удаленные зависимости..... | 620 |
| Ограничения модели удаленных вызовов Oracle..... | 622 |
| Перекомпиляция недействительных программ..... | 623 |
| Предупреждения при компиляции..... | 627 |
| Пример..... | 627 |
| Включение предупреждений компилятора..... | 628 |
| Некоторые полезные предупреждения..... | 629 |
| Тестирование программ PL/SQL..... | 636 |
| Типичные неэффективные технологии тестирования..... | 637 |
| Общие рекомендации по тестированию кода PL/SQL..... | 639 |
| Средства автоматизации тестирования программ PL/SQL..... | 640 |
| Трассировка кода PL/SQL..... | 641 |
| DBMS_UTILITY.FORMAT_CALL_STACK..... | 642 |
| UTL_CALL_STACK (Oracle Database 12c)..... | 643 |
| DBMS_APPLICATION_INFO..... | 646 |
| Трассировка с использованием opp_trace..... | 647 |
| Пакет DBMS_TRACE..... | 648 |
| Отладка программ PL/SQL..... | 651 |
| Неправильные способы организации отладки..... | 651 |
| Полезные советы и стратегии отладки..... | 653 |
| «Белые списки» и управление доступом к программным модулям..... | 657 |
| Защита хранимого кода..... | 658 |
| Ограничения..... | 659 |
| Использование программы wgar..... | 659 |
| Динамическое сокрытие кода с использованием DBMS_DDL..... | 659 |
| Работа со скрытым кодом..... | 660 |
| Знакомство с оперативной заменой (Oracle11g Release 2)..... | 661 |
| Глава 21. Оптимизация приложений PL/SQL | 664 |
| Средства, используемые при оптимизации..... | 665 |
| Анализ использования памяти..... | 665 |
| Выявление «узких мест» в коде PL/SQL..... | 665 |
| Хронометраж..... | 670 |
| Выбор самой быстрой программы..... | 671 |
| Предотвращение заикливания..... | 672 |
| Предупреждения, относящиеся к производительности..... | 673 |
| Оптимизирующий компилятор..... | 673 |
| Как работает оптимизатор..... | 675 |
| Оптимизация циклов выборки данных..... | 677 |
| Кэширование данных..... | 678 |
| Пакетное кэширование..... | 679 |
| Кэширование детерминированных функций..... | 683 |
| Кэширование результатов функций (Oracle Database 11g)..... | 685 |
| Сводка способов кэширования..... | 697 |
| Массовая обработка..... | 698 |
| Ускорение выборки с использованием BULK COLLECT..... | 699 |
| Быстрое выполнение операций DML и команда FORALL..... | 704 |

| | |
|--|------------|
| Повышение производительности с использованием конвейерных табличных функций..... | 712 |
| Замена вставки на базе строк загрузкой на базе конвейерных функций | 713 |
| Конвейерные функции при оптимизации операций слияния | 719 |
| Параллельные конвейерные функции при асинхронной выгрузке данных | 721 |
| Влияние группировки и режима потоковой передачи в параллельных конвейерных функциях..... | 724 |
| Конвейерные функции и затратный оптимизатор | 725 |
| Конвейерные функции при загрузке сложных данных | 730 |
| В завершение о конвейерных функциях..... | 736 |
| Специализированные приемы оптимизации | 736 |
| Метод передачи параметров NOCOPY | 736 |
| Выбор типа данных | 739 |
| Оптимизация вызовов функций в SQL (версия 12.1 и выше)..... | 740 |
| Общие замечания о производительности..... | 741 |
| Глава 22. Ввод/вывод в PL/SQL | 742 |
| Вывод информации | 742 |
| Включение DBMS_OUTPUT | 743 |
| Запись в буфер | 743 |
| Чтение содержимого буфера..... | 744 |
| Чтение и запись файлов..... | 745 |
| Параметр UTL_FILE_DIR..... | 745 |
| Работа с каталогами в Oracle | 746 |
| Открытие файлов..... | 748 |
| Проверка открытия файла..... | 749 |
| Закрытие файла..... | 749 |
| Чтение из файла | 749 |
| Запись в файл | 752 |
| Копирование файлов | 754 |
| Удаление файлов | 754 |
| Переименование и перемещение файлов | 755 |
| Получение атрибутов файла | 755 |
| Отправка электронной почты..... | 756 |
| Предварительная настройка | 757 |
| Настройка сетевой безопасности..... | 757 |
| Отправка короткого текстового сообщения | 758 |
| Включение «удобных» имен в адреса электронной почты..... | 760 |
| Отправка текстового сообщения произвольной длины | 760 |
| Отправка сообщения с коротким вложением | 761 |
| Отправка небольшого файла во вложении | 763 |
| Вложение файла произвольного размера..... | 764 |
| Работа с данными в Интернете (HTTP) | 765 |
| Фрагментная загрузка страницы..... | 766 |
| Загрузка страницы в объект LOB..... | 767 |
| Аутентификация HTTP | 768 |
| Загрузка зашифрованной страницы (HTTPS)..... | 769 |
| Передача данных методами GET и POST | 770 |

| | |
|--|-----|
| Запрет и долгосрочное хранение cookie..... | 773 |
| Загрузка данных с сервера FTP | 773 |
| Использование прокси-сервера | 774 |
| Другие разновидности ввода/вывода в PL/SQL | 774 |
| Каналы, очереди и оповещения..... | 774 |
| Сокеты TCP | 775 |
| Встроенный веб-сервер Oracle | 775 |

ЧАСТЬ VI. ОСОБЫЕ ВОЗМОЖНОСТИ PL/SQL

| | |
|--|------------|
| Глава 23. Безопасность и PL/SQL | 778 |
| Общие сведения о безопасности | 778 |
| Шифрование | 779 |
| Длина ключа | 781 |
| Алгоритмы | 781 |
| Заполнение и сцепление..... | 782 |
| Пакет DBMS_CRYPTO | 783 |
| Алгоритмы | 783 |
| Заполнение и сцепление..... | 783 |
| Шифрование данных | 784 |
| Шифрование LOB..... | 786 |
| SecureFiles..... | 787 |
| Дешифрование данных | 787 |
| Генерирование ключей | 788 |
| Управление ключами | 789 |
| Криптографическое хеширование | 793 |
| Коды MAC | 795 |
| Прозрачное шифрование данных | 796 |
| Прозрачное шифрование табличного пространства..... | 798 |
| Безопасность уровня строк..... | 800 |
| Зачем изучать RLS? | 802 |
| Простой пример использования RLS | 803 |
| Статические и динамические политики | 806 |
| Контекстная политика | 809 |
| Использование столбцовой модели RLS | 810 |
| Отладка RLS | 813 |
| Контексты приложений | 816 |
| Использование контекстов приложений..... | 817 |
| Безопасность в контекстах..... | 818 |
| Контексты как предикаты в RLS | 818 |
| Идентификация сторонних пользователей | 821 |
| Детализированный аудит | 823 |
| Зачем изучать FGA?..... | 824 |
| Простой пример..... | 825 |
| Количество столбцов | 826 |
| Просмотр журнала аудита | 827 |
| Подставляемые параметры | 828 |
| Модули-обработчики..... | 829 |

| | |
|---|------------|
| Глава 24. Архитектура PL/SQL..... | 831 |
| DIANA..... | 831 |
| Как Oracle выполняет код PL/SQL?..... | 832 |
| Пример..... | 832 |
| Ограничения компилятора..... | 835 |
| Пакеты по умолчанию..... | 835 |
| Модели разрешений..... | 838 |
| Модель разрешений создателя..... | 838 |
| Модель разрешений вызывающего..... | 842 |
| Комбинированная модель разрешений..... | 844 |
| Назначение ролей программам PL/SQL (Oracle Database 12c)..... | 844 |
| Функции «Кто меня вызвал?» (Oracle Database 12c)..... | 847 |
| BEQUEATH CURRENT_USER для представлений (Oracle Database 12c)..... | 847 |
| Ограничение привилегий в модели прав вызывающего (Oracle Database 12c)..... | 849 |
| Условная компиляция..... | 850 |
| Примеры условной компиляции..... | 851 |
| Директива получения информации..... | 852 |
| Директива \$IF..... | 855 |
| Директива \$ERROR..... | 856 |
| Синхронизация кода с использованием пакетных констант..... | 856 |
| Применение директив получения информации для определения конфигурации конкретных программ..... | 857 |
| Работа с обработанным кодом..... | 858 |
| PL/SQL и память экземпляров базы данных..... | 859 |
| SGA, PGA и UGA..... | 859 |
| Курсоры и память..... | 860 |
| Советы по экономии памяти..... | 861 |
| Что делать при нехватке памяти..... | 870 |
| Компиляция в низкоуровневый код..... | 873 |
| Когда используется режим интерпретации..... | 873 |
| Когда используется низкоуровневый режим..... | 873 |
| Низкоуровневая компиляция и версии Oracle..... | 873 |
| Что необходимо знать..... | 874 |
| Глава 25. Глобализация и локализация в PL/SQL | 876 |
| Общие сведения и терминология..... | 877 |
| Знакомство с Юникодом..... | 878 |
| Типы данных и национальные наборы символов..... | 880 |
| Кодировка символов..... | 881 |
| Параметры Globalization Support (NLS)..... | 882 |
| Функции Юникода..... | 882 |
| Символьная семантика..... | 887 |
| Порядок сортировки строк..... | 890 |
| Двоичная сортировка..... | 891 |
| Одноязычная сортировка..... | 891 |
| Многоязычная сортировка..... | 893 |
| Многоязыковой информационный поиск..... | 894 |
| Информационный поиск и PL/SQL..... | 896 |

| | |
|--|------------|
| Дата и время | 899 |
| Типы данных временных меток | 899 |
| Форматирование даты и времени | 900 |
| Преобразования денежных величин | 902 |
| Globalization Development Kit для PL/SQL | 904 |
| Пакет UTL_118N | 904 |
| Пакет обработки ошибок UTL_LMS | 907 |
| Варианты реализации GDK | 908 |
| Глава 26. Объектно-ориентированные возможности PL/SQL | 910 |
| История объектных возможностей Oracle | 910 |
| Пример объектного приложения | 912 |
| Создание базового типа | 913 |
| Создание подтипа | 914 |
| Методы | 915 |
| Вызов методов супертипа в Oracle11g | 918 |
| Запись, выборка и использование объектов | 920 |
| Эволюция и создание типов | 926 |
| И снова указатели | 928 |
| Типы данных ANY | 933 |
| Сделай сам | 936 |
| Сравнение объектов | 939 |
| Объектные представления | 943 |
| Пример реляционной системы | 945 |
| Объектное представление с атрибутом-коллекцией | 946 |
| Объектные подпредставления | 948 |
| Объектное представление с обратным отношением | 950 |
| Триггеры INSTEAD OF | 950 |
| Различия между объектными представлениями и объектными таблицами | 952 |
| Сопровождение объектных типов и объектных представлений | 953 |
| Словарь данных | 953 |
| Привилегии | 954 |
| О целесообразности применения объектно-ориентированного подхода | 956 |
| Глава 27. Вызов Java-программ из PL/SQL | 958 |
| Oracle и Java | 958 |
| Подготовка к использованию Java в Oracle | 960 |
| Установка Java | 960 |
| Построение и компиляция кода Java | 960 |
| Назначение разрешений для разработки и выполнения Java-кода | 961 |
| Простая демонстрация | 963 |
| Поиск функциональности Java | 964 |
| Построение класса Java | 964 |
| Компиляция и загрузка в Oracle | 966 |
| Построение обертки PL/SQL | 967 |
| Удаление файлов из PL/SQL | 968 |
| Использование loadjava | 968 |

| | |
|--|-------------|
| Программа dropjava | 970 |
| Управление Java в базе данных | 971 |
| Пространство имен Java в Oracle..... | 971 |
| Получение информации о загруженных элементах Java | 971 |
| Пакет DBMS_JAVA | 972 |
| LONGNAME: преобразование длинных имен Java..... | 972 |
| GET_, SET_ и RESET_COMPILER_OPTION: чтение и запись параметров компилятора..... | 973 |
| SET_OUTPUT: включение вывода из Java | 974 |
| EXPORT_SOURCE, EXPORT_RESOURCE и EXPORT_CLASS: экспортирование объектов схемы | 974 |
| Публикация и использование кода Java в PL/SQL | 976 |
| Спецификация вызова | 976 |
| Правила спецификаций вызовов..... | 977 |
| Отображение типов данных | 978 |
| Вызов метода Java в SQL..... | 979 |
| Обработка исключений в Java | 979 |
| Расширение функциональности файлового ввода/вывода | 981 |
| Другие примеры | 985 |
| Глава 28. Внешние процедуры | 987 |
| Знакомство с внешними процедурами | 988 |
| Пример: вызов команды операционной системы | 988 |
| Архитектура внешних процедур..... | 990 |
| Конфигурация Oracle Net | 991 |
| Определение конфигурации слушателей..... | 991 |
| Характеристики безопасности | 993 |
| Настройка многопоточного режима..... | 994 |
| Создание библиотеки Oracle..... | 996 |
| Написание спецификации вызова..... | 997 |
| Общий синтаксис спецификации вызова..... | 998 |
| Снова об отображении параметров..... | 999 |
| Отображение параметров: полная картина..... | 1001 |
| Секция PARAMETERS..... | 1002 |
| Свойства PARAMETERS | 1003 |
| Инициирование исключений из вызываемой программы C | 1005 |
| Нестандартные агенты | 1007 |
| Сопровождение внешних процедур..... | 1010 |
| Удаление библиотек | 1010 |
| Словарь данных | 1010 |
| Правила и предупреждения | 1010 |
| Приложение А. Параметры функций и метасимволы регулярных выражений | 1012 |
| Приложение Б. Числовые форматы | 1017 |
| Приложение В. Маска формата даты | 1020 |

Предисловие

Миллионы разработчиков приложений и администраторов баз данных во всем мире используют программный продукт, созданный Oracle Corporation, для построения сложных систем, работающих с огромными объемами данных. В основе большей части Oracle лежит PL/SQL — язык программирования, который предоставляет процедурные расширения используемой в Oracle версии SQL, а также служит языком программирования инструментария Oracle Developer (в первую очередь Forms Developer и Reports Developer).

PL/SQL выходит на первый план почти во всех новых продуктах Oracle Corporation. Профессиональные программисты используют PL/SQL для решения самых разнообразных задач, среди которых:

- реализация основной бизнес-логики в Oracle Server в виде хранимых процедур и триггеров базы данных;
- генерирование документов XML и управление ими исключительно на уровне базы данных;
- связывание веб-страниц с базой данных Oracle;
- реализация и автоматизация задач администрирования базы данных — от формирования низкоуровневой системы безопасности до управления сегментами отката в программах PL/SQL.

Язык PL/SQL создавался по образцу Ada¹ — языка программирования, разработанного для Министерства обороны США. Высокоуровневый язык Ada уделяет особое внимание абстракции данных, сокрытию информации и другим ключевым элементам современных стратегий проектирования. Благодаря грамотным архитектурным решениям PL/SQL стал мощным языком, в котором реализованы многие современные элементы процедурных языков:

- Полный диапазон типов данных, от чисел и строк до сложных структур данных (например, записей — аналогов строк в реляционных таблицах), коллекций (аналоги массивов в Oracle) и XMLType (тип для работы с документами XML в Oracle и средствами PL/SQL).
- Четкая, удобочитаемая блочная структура, упрощающая расширение и сопровождение приложений PL/SQL.
- Условные, циклические и последовательные управляющие конструкции, включая CASE и три разных вида циклов.
- Обработчики исключений для обработки ошибок на базе событий.
- Именованные, пригодные для повторного использования элементы кода: функции, процедуры, триггеры, объектные типы (аналоги классов в объектно-ориентированном программировании) и пакеты (группы логически связанных программ и переменных).

Язык PL/SQL плотно интегрирован в Oracle SQL: SQL-инструкции могут выполняться прямо из процедурных программ без использования промежуточных интерфейсов вроде JDBC (Java Database Connectivity) или ODBC (Open Database Connectivity). И наоборот, функции PL/SQL могут вызываться из SQL-инструкций.

Разработчики Oracle-приложений, желающие добиться успеха в XXI веке, должны научиться в полной мере использовать возможности PL/SQL. Этот процесс состоит

¹ Язык получил свое название в честь математика Ады Лавлейс, которую многие считают первым программистом в мире.

из двух шагов. Во-первых, вы должны освоить постоянно расширяющийся набор возможностей языка, а во-вторых, после изучения отдельных возможностей необходимо научиться объединять все эти конструкции для построения сложных приложений.

По этим и другим причинам разработчику Oracle необходим надежный, исчерпывающий источник информации об основных возможностях языка PL/SQL. Он должен знать основные структурные элементы PL/SQL и иметь под рукой качественные примеры, чтобы избежать многочисленных проб и ошибок. Как и в любом языке программирования, в PL/SQL у каждой задачи есть правильное решение и много неправильных (или по крайней мере «менее правильных») решений. Надеемся, эта книга поможет вам в эффективном изучении PL/SQL.

Основные цели

Итак, в решении каких же конкретных задач вам поможет эта книга?

- **Изучение всех возможностей PL/SQL.** В справочной документации Oracle описаны все возможности языка PL/SQL, но в ней не говорится о том, как применять эти технологии на практике. Книги и учебные курсы обычно излагают один и тот же стандартный материал. Эта книга выходит за рамки основных возможностей языка и демонстрирует нестандартные способы использования тех или иных возможностей для достижения нужного результата.
- **Применение PL/SQL для решения ваших задач.** Разработчик не проводит дни и ночи за написанием модулей PL/SQL, которые помогут ему достичь просветления. Язык PL/SQL предназначен для решения конкретных задач вашей организации или ваших клиентов. В этой книге я постараюсь показать, как решаются реальные задачи — те, с которыми программисты сталкиваются в повседневной работе. Поэтому здесь приведено так много примеров, причем не только небольших фрагментов кода, но и достаточно крупных компонентов приложений, которые, возможно с определенными модификациями, читатели смогут интегрировать в собственные проекты. Очень много кода содержится в самой книге, но еще больше его содержится на сайте издательства O'Reilly. Многие примеры демонстрируют аналитический процесс выработки эффективных решений. Вы поймете, как на практике применять предлагаемые средства PL/SQL, пользуясь нестандартными или недокументированными возможностями и методами.
- **Создание эффективного кода, который легко сопровождать.** PL/SQL и другие продукты Oracle обладают большим потенциалом, позволяющим в рекордно короткие сроки разрабатывать достаточно мощные приложения. Но если не проявить должной осторожности, этот потенциал только навлечет на вас большие неприятности. Если бы эта книга только научила вас писать в предельно короткие сроки необходимые программы, мы бы не посчитали свою задачу выполненной. Я хочу помочь вам освоить принципы и приобрести навыки построения гибких приложений, программный код которых легко читается, понятен, без труда модифицируется — словом, требует минимальных затрат на сопровождение и доработку. Я хочу научить вас применять полноценные стратегии и программные архитектуры, чтобы вы находили мощные и универсальные способы применения PL/SQL для решения ваших задач.

Структура книги

Авторы и O'Reilly Media приложили максимум усилий, чтобы как можно более полно осветить процесс развития PL/SQL. В шестом издании книги описываются средства

и возможности PL/SQL для Oracle12c Release 1; эта версия будет считаться «эталонной». Однако там, где это уместно, в книге будут упоминаться другие возможности, появившиеся в более ранних версиях (или доступные только в них). Список основных характеристик разных версий приведен в разделе «История PL/SQL» главы 1. Язык PL/SQL прошел долгий путь с момента выхода версии 1.0 для Oracle 6. В книгу также были внесены серьезные изменения, посвященные новшествам PL/SQL, и был добавлен новый материал.

Главным изменением шестого издания является подробное описание всех новых возможностей PL/SQL в Oracle Database 12c Release 1. Сводка этих нововведений приведена в главе 1 со ссылками на главы, в которых эти нововведения рассматриваются подробно.

Я очень доволен результатами своего труда и надеюсь, что ими будете довольны и вы. Книга содержит больше информации, чем любое из предыдущих изданий, но как мне кажется, нам удалось сохранить чувство юмора и разговорный стиль изложения, благодаря которому, как мне сообщали читатели за прошедшие годы, эта книга легко читалась, была понятной и полезной.

И одно замечание по поводу авторского «голоса» за текстом. Вероятно, вы заметите, что в одних частях книги используется слово «мы», а в других — «я». Одной из особенностей книги (которая, кстати, понравилась читателям) стал персонализированный стиль изложения, неотделимый от текста. Соответственно, даже с появлением соавторов (а в третьем, четвертом и пятом изданиях — значительном творческом вкладе от других людей) мы решили сохранить использование «я» там, где автор говорит от своего имени.

А чтобы вам не пришлось гадать, какой из основных авторов стоит за «я» в каждой главе, мы приведем краткий перечень для любознательных. Дополнительная информация о других участниках приводится в разделе «Благодарности».

| Глава | Автор | Глава | Автор |
|-------------|------------------------|-------|--------------------|
| Предисловие | Стивен | 15 | Стивен |
| 1 | Стивен | 16 | Стивен |
| 2 | Билл и Стивен | 17 | Стивен |
| 3 | Стивен и Билл | 18 | Стивен |
| 4 | Стивен, Чип и Джонатан | 19 | Даррил и Стивен |
| 5 | Стивен и Билл | 20 | Стивен |
| 6 | Стивен | 21 | Стивен и Адриан |
| 7 | Чип, Джонатан и Стивен | 22 | Билл и Стивен |
| 8 | Чип, Джонатан и Стивен | 23 | Аруп |
| 9 | Чип, Джонатан и Стивен | 24 | Билл, Стивен и Чип |
| 10 | Чип, Джонатан и Стивен | 25 | Рон |
| 11 | Стивен | 26 | Билл и Стивен |
| 12 | Стивен и Билл | 27 | Билл и Стивен |
| 13 | Чип и Джонатан | 28 | Билл и Стивен |
| 14 | Стивен | | |

О содержании

Шестое издание книги состоит из шести частей.

- **Часть I. «Программирование на PL/SQL».** Глава 1 начинается с самого начала: как появился SQL? Для чего он нужен? Далее приводится краткий обзор основных возможностей PL/SQL. Глава 2 построена таким образом, чтобы вы могли сходу взяться

за программирование: она содержит четкие и простые инструкции по выполнению кода PL/SQL в среде разработки SQL*Plus и некоторых других распространенных средах. В главе 3 рассказывается об основах языка PL/SQL: что собой представляют операторы, какова структура блока, как создавать комментарии и т. п.

- **Часть II. «Структура программы PL/SQL».** В главах 4–6 рассматриваются условные (IF и CASE) и последовательные (GOTO и NULL) операторы управления порядком выполнения команд; циклы и оператор CONTINUE, появившийся в Oracle11; средства обработки исключений в PL/SQL. Эта часть книги учит составлять блоки программного кода, соответствующие сложным требованиям ваших приложений.
- **Часть III. «Работа с данными в PL/SQL».** Почти любая написанная вами программа будет заниматься обработкой данных, которые часто являются локальными для процедуры или функции PL/SQL. Главы 7–13 посвящены различным типам программных данных, определяемым непосредственно в PL/SQL: числам, строкам, датам, временным меткам, записям и коллекциям. Вы узнаете о новых типах данных Oracle11g (SIMPLE_INTEGER, SIMPLE_FLOAT и SIMPLE_DOUBLE), а также о многих типах для работы с двоичными данными, датой и временем, введенных в других версиях. Кроме того, мы расскажем о встроенных функциях, предоставляемых Oracle для выполнения различных операций с данными.
- **Часть IV. «SQL и PL/SQL».** В главах 14–16 рассказано об одном из центральных аспектов программирования на PL/SQL: подключении к базе данных, осуществляемом из кода SQL. Из них вы узнаете, как определяются транзакции обновления, вставки, слияния и удаления таблиц базы данных; как запросить из базы данных информацию для обработки в программах PL/SQL и как динамически выполнять SQL-инструкции средствами NDS (Native Dynamic SQL).
- **Часть V. «Создание приложений PL/SQL».** В этой части книги сводится воедино все, о чем говорилось ранее. Приступая к ее изучению, вы уже будете знать, как объявлять переменные и как с ними работать, освоите важнейшие принципы обработки ошибок и построения циклов. В главах 17–22 рассказывается о самых крупных структурных элементах приложений: процедурах, функциях, триггерах и пакетах, а также об организации ввода и вывода информации в приложениях PL/SQL. В главе 20 также обсуждаются вопросы управления кодом PL/SQL, его тестирования, отладки и управления зависимостями; также здесь представлен обзор механизма оперативной замены, введенного в Oracle11g Release 2. Глава 21 посвящена использованию различных инструментов и приемов для достижения оптимального быстродействия в программах PL/SQL. В главе 22 описаны средства ввода/вывода PL/SQL, от пакетов DBMS_OUTPUT (вывода на экран) и UTL_FILE (чтение и запись файлов) до UTL_MAIL (отправка электронной почты) и UTL_HTTP (получение данных с веб-страниц).
- **Часть VI. «Особые возможности PL/SQL».** Язык PL/SQL, столь мощный и богатый, содержит немало функциональных возможностей и структурных элементов, которые не используются в повседневной работе, но позволяют максимально просто и эффективно решать задачи, справиться с которыми другими способами было бы очень трудно. В главе 23 описаны проблемы безопасности, с которыми мы сталкиваемся при создании программ PL/SQL. В главе 24 рассматривается архитектура PL/SQL, в том числе использование памяти. Глава 25 содержит полезный материал для разработчиков, которым необходимо решать проблемы глобализации и локализации в своих приложениях. Глава 26 содержит вводный курс по объектно-ориентированным возможностям Oracle (объектным типам и представлениям).

В приложениях А–В содержится информация о синтаксисе регулярных выражений, форматах чисел и дат.

Главы, посвященные выполнению кода Java и C из приложений PL/SQL, входившие в печатное четвертое издание, были перемещены на веб-сайт книги.

Начинающему программисту PL/SQL стоит прочитать книгу от начала до конца — она повысит вашу квалификацию и углубит знания PL/SQL. Опытным разработчикам PL/SQL, возможно, будет достаточно ознакомиться лишь с отдельными ее разделами, чтобы освоить конкретные приемы практического программирования. Но независимо от того, в каком качестве — учебника или справочного пособия — вы будете использовать эту книгу, надеемся, что она поможет вам научиться эффективно применять PL/SQL в своей работе.

Какие темы не рассматриваются

Даже в самой толстой книге нельзя рассказать обо всем. Oracle — огромная и сложная система, а мы рассматриваем только основные возможности языка PL/SQL. Ниже перечислены темы, которые не вошли в рамки нашего издания и поэтому лишь поверхностно упоминаются время от времени.

Язык SQL. Предполагается, что читатель уже знает язык SQL и умеет составлять инструкции SELECT, INSERT, UPDATE, MERGE и DELETE.

Администрирование баз данных Oracle. Администраторы баз данных почерпнут из этой книги немало полезного и интересного — в частности, научатся писать PL/SQL-программы для создания и обслуживания баз данных. Тем не менее у нас не было возможности рассказать о нюансах языка определения данных (DDL), входящего в Oracle SQL.

Оптимизация приложений и баз данных. Тема оптимизации также рассматривается лишь поверхностно, хотя в главе 21 обсуждаются многие приемы и инструменты, которые пригодятся вам при оптимизации быстродействия ваших программ PL/SQL.

Технологии разработки приложений Oracle, независимые от PL/SQL. Книга даже не пытается демонстрировать построение приложений с использованием таких инструментов, как Oracle Forms Developer, несмотря на то, что в них также используется язык PL/SQL. Я решил сконцентрироваться на фундаментальных возможностях языка и на тех операциях, которые с его помощью можно выполнять в базах данных. Однако большая часть материала книги относится и к использованию PL/SQL в Forms Developer и Reports Developer.

Платформа и версия

Материал книги и представленные примеры в основном не относятся к какой-либо конкретной машине и/или операционной системе. В тех случаях, когда приведенное описание зависит от версии (например, оно может использоваться только в Oracle11g или в конкретном выпуске — скажем, в Oracle11g Release 2), об этом явно говорится в тексте.

Существует множество версий PL/SQL, и не исключено, что вы будете пользоваться даже несколькими из них в своей работе. В главе 1 описаны самые распространенные версии PL/SQL и то, что вам необходимо знать о них (см. раздел «О версиях» главы 1).

О программном коде

Весь приведенный в книге программный код доступен на веб-странице издательства O'Reilly: <http://www.oreil.ly/oracle-plsql-sixth>. Кроме примеров, по указанному адресу вы найдете материал глав из предыдущих изданий книги, изъятых или значительно

сокращенных в разных изданиях книги. Они могут быть особенно полезны читателям, работающим со старыми версиями Oracle.

Информацию о всех книгах Стивена Фейерштейна и дополнительные ресурсы также можно найти по адресу: <http://www.stevenfeuerstein.com>. Также читателям будет полезно посетить сайт PL/SQL Obsession (портал Стивена Фейерштейна, посвященный PL/SQL) по адресу <http://www.ToadWorld.com/SF>. Здесь вы найдете учебные материалы, примеры кода и многое другое.

Чтобы найти код конкретного примера на сайте, ищите по имени файла, указанному в тексте. Во многих примерах оно указывается в комментарии из первой строки:

```
/* Файл в Сети: fullname.pkg */
```

Если в интересующем вас примере кода комментарий «Файл в Сети» отсутствует, обратитесь к файлу кода соответствующей главы.

В этих файлах содержатся все фрагменты и примеры, которые не заслуживают выделения в отдельный файл, но могут пригодиться для копирования/вставки. Кроме того, в них также содержатся DDL-инструкции для создания таблиц и других объектов, необходимых для работы кода.

Файл кода каждой главы называется *chNN_code.sql*, где *NN* — номер главы.

Наконец, сценарий *hr_schema_install.sql* создает стандартные демонстрационные таблицы Oracle Human Resources — такие, как *employees* и *departments*. Эти таблицы часто используются в примерах книги.

Использование примеров кода

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу <http://oreil.ly/oracle-plsql-sixth>.

Эта книга написана для того, чтобы помочь вам в решении конкретных задач. В общем случае вы можете использовать приводимые примеры в своих программах и документации. Связываться с авторами для получения разрешения не нужно, если только вы не воспроизводите значительный объем кода. С другой стороны, для продажи или распространения дисков CD-ROM с примерами из книг O'Reilly потребуются разрешения, как и в том случае, если значительный объем кода из примеров книги включается в документацию по вашему продукту. Мы будем признательны за ссылку на источник информации, хотя и не требуем ее. Если вы полагаете, что ваши потребности выходят за рамки оправданного использования примеров кода или разрешений, приведенных выше, свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

За время, прошедшее после выхода первого издания в 1995 году, эта книга прошла долгий путь, став авторитетным руководством по использованию языка PL/SQL. И за это я хочу прежде всего выразить признательность нашим читателям.

Чтобы книга содержала точную, доступную и актуальную информацию о PL/SQL, в нее был вложен огромный (да, признаюсь — порой непосильный) труд; безусловно, эта работа была бы невозможна без помощи множества других специалистов по Oracle, наших друзей и членов семей и, конечно, замечательных людей из O'Reilly Media.

Мы постарались упомянуть всех, кто помогал в работе над шестым изданием книги.

Прежде всего нам хотелось бы поблагодарить соавторов, которые внесли вклад в работу над книгой в виде целых глав и/или материалов; это Эдриан Биллингтон (Adrian

Billington), Чип Доуз (Chip Dawes), Джонатан Генник (Jonathan Gennick), Рон Хардман (Ron Hardman), Дэррил Харли (Darryl Hurley) и Аруп Нанда (Arup Nanda). В этом издании Чип Доуз взял на себя ответственность за обновление нескольких глав. Джонатан Генник написал или значительно переработал шесть глав в прошлых изданиях. Дэррил Харли обновлял главу, посвященную триггерам, в нескольких изданиях книги и поделился информацией о средствах интернационализации Oracle. Аруп Нанда написал главу, посвященную безопасности. Рон Хардман написал главу о глобализации и локализации, а Эдриан Биллингтон предоставил материал для главы 21.

Я предложил нашим соавторам написать несколько слов о себе.

Эдриан Биллингтон — консультант в области проектирования, разработки и оптимизации приложений; работает с базами данных Oracle с 1999 года. Его усилиями был создан сайт *oracle-developer.net* с описанием множества приемов и возможностей SQL и PL/SQL для разработчиков Oracle. Эдриан также является сертифицированным специалистом Oracle и участником сети OakTable Network. Он благодарит Джеймса Пэдфилда (James Padfield), Тома Кайта (Tom Kyte) и Стивена Фейерштейна, которые вдохновляли его на первых порах его становления как профессионала Oracle. Эдриан живет в Великобритании с женой Энджи и тремя дочерьми: Джорджией, Оливией и Изабеллой.

Чип Доуз занимается построением и сопровождением систем в реляционных СУБД уже больше 20 лет; основные занятия — администрирование, разработка и преподавание. В настоящее время работает менеджером в PwC, где помогает клиентам извлечь максимум пользы из их данных. Чип живет в окрестностях Чикаго с женой и детьми.

Джонатан Генник — профессионал, известный своими познаниями в области Oracle. В прошлом занимался как разработкой программного обеспечения, так и администрированием баз данных. В качестве разработчика всегда уделял особое внимание диагностике и отладке. Любит работать с SQL и PL/SQL, хорошо известен своими книгами и статьями в этой области. В свободное время Джонатан занимается делами, не связанными с техникой: он активно участвует в делах местной церкви, где его можно часто встретить за обсуждением Священного Писания со школьниками и даже студентами, а иногда даже за кафедрой. Он является заядлым любителем горного велосипеда, причем ездит даже зимой на шипованных шинах, импортированных из Финляндии. В настоящее время Джонатан занимается исследованиями в области встроенных статистических функций Oracle SQL.

Рон Хардман является основателем фирмы SettleOurEstate.com — решения по управлению недвижимостью, построенному на базе Oracle Apex и Oracle Cloud Database. Также проводит консультации в области Oracle Text и технологий глобализации Oracle по всему миру, оставаясь работником и клиентом Oracle в течение более 17 лет. Рон пишет не только на технологические темы; в 2010 году он выпустил свой первый исторический роман «Shadow Fox: Sons of Liberty», написанный совместно с дочерью.

Дэррил Харли работает с технологиями Oracle более 20 лет, уделяя основное внимание PL/SQL и администрированию. Живет в Ричмонде (Канада) с женой Ванессой и замечательной дочерью Бианкой.

Аруп Нанда работает администратором Oracle с 1993 года, занимаясь всеми аспектами администрирования — моделированием, оптимизацией, программированием PL/SQL, архивацией, восстановлением после сбоев и многими другими. Работает ведущим администратором баз данных в крупной корпорации, написал более 500 статей, участвовал в написании пяти книг, выступал с докладами на 300 с лишним конференциях. Занимается преподаванием, участвует в специальных проектах и пишет о технологиях Oracle в своем блоге *arup.blogspot.com*. Был избран «Администратором года» журналом Oracle

Magazine в 2003 году и «Специалистом по архитектурам года» в 2012 году, является обладателем сертификата OCP, директором OTN ACT и участником сети OakTable Network. Живет в Коннектикуте, США, с женой Ану и сыном Анишем.

Для рецензирования такой большой книги потребовалось немало специалистов, причем мы просили их протестировать каждый фрагмент кода и каждую программу, чтобы в печатной версии осталось как можно меньше ошибок. Мы глубоко признательны профессионалам по Oracle PL/SQL, которые не жалели собственного времени для совершенствования нашего труда.

За работу над шестым изданием я хочу прежде всего поблагодарить Валентина Никотина (Valentin Nikotin) — одного из лучших научных редакторов, работавших над этой книгой. Он не только проверил правильность материала Oracle Database 12c, но и помог с устранением неоднозначностей и исправлением ошибок в других ключевых главах, оставшихся неизменными в этом издании. Мои научные редакторы сильно повлияли на качество книги. Огромное спасибо Патрику Барелу (Patrick Barel) и Арупу Нанде!

Выражаем глубокую признательность Брину Луэллину (Bryn Llewellyn), менеджеру продукта PL/SQL, и другим членам группы разработки PL/SQL — прежде всего, Чарльзу Уэзереллу (Charles Wetherell). Брин предоставил важную информацию о новых функциях Oracle 12c, отвечал на наши бесконечные вопросы о различных особенностях языка PL/SQL, проявляя при этом безграничное терпение. Несомненно, своим пониманием PL/SQL и точностью изложения материала мы во многом обязаны Брину.

В том, что не относится к Oracle, я глубоко благодарен Джоэлу Финкелю (Joel Finkel) — моему любимому «универсалу», компенсирующему узкую специализацию, которая одновременно помогает и ограничивает мои способности в том, что касается компьютеров и программного обеспечения.

Конечно, речь шла только о технической стороне дела. Когда мы почувствовали, что материал PL/SQL «доведен до ума», команда специалистов из O'Reilly Media во главе с Энн Спенсер (Ann Spencer) сделала все возможное, чтобы книга получилась достойной такого уважаемого издательства, как O'Reilly. Огромная благодарность Джулии Стил (Julie Steele), Николь Шелби (Nicole Shelby), Робу Романо (Rob Romano) и всем остальным. Это первое издание, редактором которого была Энн. Во всех предыдущих изданиях (с 1994 до 2007 года) мы имели честь и удовольствие работать с Дебби Рассел (Debby Russell). Спасибо, Дебби, за многолетнюю работу, направленную на успех серии Oracle в O'Reilly Media!

Есть еще немало других людей, которых мы благодарили (и продолжаем благодарить) за вклад в первые пять изданий книги: Сохаиб Абасси (Sohaib Abassi), Стив Адамс (Steve Adams), Дон Бейлз (Don Bales), Кайлин Баркли (Cailein Barclay), Патрик Барел (Patrick Barel), Джон Бересневич (John Beresiewicz), Том Бертофф (Tom Berthoff), Сунил Бхаргава (Sunil Bhargava), Дженнифер Блер (Jennifer Blair), Дик Болз (Dick Bolz), Брайан Боултон (Bryan Boulton), Пер Брондум (Per Brondum), Борис Берштейн (Boris Burshteyn), Эрик Кемплин (Eric Camplin), Джо Селко (Joe Celko), Гэри Керносек (Gary Cernosek), Барри Чейз (Barry Chase), Джефф Честер (Geoff Chester), Айвен Чонг (Ivan Chong), Дэн Кламадж (Dan Clamage), Грей Клоссман (Gray Clossman), Эвери Коэн (Avery Cohen), Роберт А. Дж. Кук (Robert A. G. Cook), Джон Корделл (John Cordell), Стив Коснер (Steve Cosner), Тони Кроуфорд (Tony Crawford), Дэниел Кронк (Daniel Cronk), Эрван Дарнелл (Ervan Darnell), Лекс де Хаан (Lex de Haan), Томас Данбер (Thomas Dunbar), Билл Дуайт (Bill Dwight), Стив Эрлих (Steve Ehrlich), Ларри Элкинс (Larry Elkins), Брюс Эпстейн (Bruce Epstein), Джоэл Финкел (Joel Finkel), Р. Джеймс Форсайт (R. James Forsythe), Майк Ганглер (Mike Gangler), Беверли Гибсон (Beverly Gibson),

Стив Гиллис (Steve Gillis), Эрик Гивлер (Eric Givler), Рик Гринуолд (Rick Greenwald), Радхакришна Хари (Radhakrishna Hari), Джерард Хартгерс (Gerard Hartgers), Дональд Херкимер (Donald Herkimer), Стив Хилкер (Steve Hilker), Билл Хинман (Bill Hinman), Гэбриел Хоффман (Gabriel Hoffman), Чандрасекаран Айер (Chandrasekharan Iyer), Кен Джейкобс (Ken Jacobs), Хакан Якобссон (Hakan Jakobsson), Джованни Харамильо (Giovanni Jaramillo), Дуэйн Кинг (Dwayne King), Марсел Кратовчил (Marcel Kratochvil), Томас Куриан (Thomas Kurian), Том Кайт (Tom KYTE), Бен Линдси (Ben Lindsey), Питер Линсли (Peter Linsley), Вадим Лоевски (Vadim Loevski), Лео Лок (Leo Lok), Дебра Луик (Debra Luik), Джеймс Мэллори (James Mallory), Радж Маттамал (Raj Mattamal), Нимиш Мехта (Nimish Mehta), Ари Мозес (Ari Mozes), Стив Манч (Steve Muench), Джефф Мюллер (Jeff Muller), Каннан Мутхукарруппан (Kannan Muthukkaruppan), Дэн Норрис (Dan Norris), Алекс Нуйтен (Alex Nuijten), Джеймс Пэдфилд (James Padfield), Ракеш Пател (Rakesh Patel), Карен Пейзер (Karen Peiser), Фред Полизо (Fred Polizo), Дэйв Познер (Dave Posner), Патрик Прибыл (Patrick Pribyl), Нэнси Прист (Nancy Priest), Шириш Пураник (Shirish Puranik), Крис Рейсикот (Chris Racicot), Шри Раджан (Sri Rajan), Марк Рихтер (Mark Richter), Крис Риммер (Chris Rimmer), Алекс Романкевич (Alex Romankevich), Берг Скалзо (Bert Scalzo), Пит Шаффер (Pete Schaffer), Скотт Соуэрс (Scott Sowers), Джей-Ти Томас (JT Thomas), Дэвид Томпсон (David Thompson), Эдвард Ван Хаттен (Edward Van Hatten), Питер Вастерд (Peter Vasterd), Андре Вергисон (Andre Vergison), Марк Вилрокс (Mark Vilroks), Зона Уолкотт (Zona Walcott), Билл Уоткинс (Bill Watkins), Чарльз Уэзерелл (Charles Wetherell), Эдвард Уайлз (Edward Wiles), Дэниел Вонг (Daniel Wong), Соломон Якобсон (Solomon Jakobson), Минь Ху Янь (Ming Hui Yang) и Тони Зимба (Tony Ziemba).

Напоследок я выражаю глубокую благодарность моей жене Ве́ве Сильва, которая поддерживала меня на каждом шаге карьеры в мире программирования. Мои мальчики Кристофер Таварес Сильва и Эли Сильва Фейерштейн хорошо перенесли недостаток моего внимания, которое было нацелено на PL/SQL (а когда были подростками — даже радовались этому). Наконец, я хочу поблагодарить Криса и его прелестную, умную и талантливую жену Лорен за мою первую внучку, Лою Люсиль Сильву.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.



Программирование на PL/SQL

В первой части книги читатель познакомится с языком PL/SQL, научится создавать и выполнять код PL/SQL и освоит основные концепции языка. Глава 1 отвечает на важнейшие вопросы: как появился язык PL/SQL? для каких задач он был создан? каковы его отличительные особенности? Глава 2 поможет вам как можно быстрее научиться писать и запускать простейшие программы. В ней приводятся четкие, понятные инструкции по выполнению кода PL/SQL в SQL*Plus и некоторых других средах. Глава 3 посвящена структуре программ и ключевым словам языка: из каких частей состоят команды PL/SQL? что такое блок PL/SQL? как включить комментарий в программу?

1

Введение в PL/SQL

PL/SQL — это сокращение от «Procedural Language extensions to the Structured Query Language», что в переводе с английского означает «процедурные языковые расширения для SQL». SQL — повсеместно распространенный язык для выборки и обновления информации (вопреки названию) в реляционных базах данных. Компания Oracle Corporation разработала PL/SQL для преодоления некоторых ограничений SQL, а также для того, чтобы предоставить более полное совершенное решение для разработчиков ответственных приложений баз данных Oracle. В этой главе читатель познакомится с PL/SQL, его происхождением и разными версиями. В ней приведена краткая сводка PL/SQL в последней версии Oracle, Oracle Database 12c. Глава завершается списком дополнительных ресурсов для разработчиков PL/SQL и некоторыми рекомендациями.

Что такое PL/SQL?

Язык PL/SQL обладает следующими определяющими характеристиками.

- **Высокая структурированность, удобочитаемость и доступность.** Новичок сможет легко постигнуть азы своей профессии с PL/SQL — этот язык прост в изучении, а его ключевые слова и структура четко выражают смысл кода. Программист с опытом работы на других языках очень быстро привыкнет к новому синтаксису.
- **Стандартный переносимый язык разработки приложений для баз данных Oracle.** Если вы написали на PL/SQL процедуру или функцию для базы данных Oracle, находящейся на портативном компьютере, то эту же процедуру можно будет перенести в базу данных на компьютере корпоративной сети и выполнить ее без каких-либо изменений (конечно, при условии совместимости версий Oracle). «Написать один раз и использовать везде» — этот основной принцип PL/SQL был известен задолго до появления языка Java. Впрочем, «везде» в данном случае означает «при работе с любой базой данных Oracle».
- **Встроенный язык.** PL/SQL не используется как самостоятельный язык программирования. Это встроенный язык, работающий только в конкретной управляющей среде. Таким образом, программы PL/SQL можно запускать из базы данных (скажем, через интерфейс SQL*Plus). Также возможно определение и выполнение программ PL/SQL из формы или отчета Oracle Developer (*клиентский PL/SQL*). Однако вы не сможете создать исполняемый файл программы на PL/SQL и запускать его автономно.
- **Высокопроизводительный, высокоинтегрированный язык баз данных.** В настоящее время существует много способов написания программ, работающих с базами данных Oracle. Например, можно использовать Java и JDBC или Visual Basic

и ODBC, а можно воспользоваться, скажем, Delphi, C++ и т. д. Однако эффективный код для работы с базой данных Oracle проще написать на PL/SQL, чем на любом другом языке программирования. В частности, Oracle имеет несколько расширений, предназначенных специально для PL/SQL, таких как инструкция `FORALL`, позволяющая повысить производительность обработки запросов на порядок и более.

История PL/SQL

В отрасли программного обеспечения компания Oracle является лидером в использовании декларативного, непроведурного подхода к проектированию баз данных и приложений. Технология Oracle Server считается одной из самых прогрессивных, мощных и надежных реляционных баз данных в мире. Средства разработки приложений от Oracle (такие, как Oracle Forms) обеспечивают высокую производительность за счет применения визуального проектирования — подхода, в котором многие характеристики программ и их элементов определяются по умолчанию, что избавляет программиста от огромного объема рутинной работы.

Истоки PL/SQL

Вначале Oracle-разработчиков в полной мере удовлетворял декларативный подход SQL в сочетании с новаторской реляционной технологией. Но с развитием отрасли возрастали и требования к средствам разработки. Все чаще разработчики желали «проникнуть вовнутрь» продуктов. Им нужно было встраивать в свои формы и сценарии достаточно сложные формулы, исключения и правила.

Выпущенная в 1988 году версия Oracle 6 стала важным шагом в развитии технологии баз данных Oracle. Ключевым компонентом новой версии стало так называемое «процедурное дополнение», или PL/SQL. Примерно в то же время появилось долгожданное обновление SQL*Forms версии 2.3. Сейчас этот продукт называется Oracle Forms или Forms Developer). В SQL*Forms 3.0 был впервые интегрирован язык PL/SQL, позволяющий разработчику просто и естественно программировать процедурную логику.

Возможности первой версии PL/SQL были весьма ограниченными. На стороне сервера этот язык использовался только для написания сценариев «пакетной обработки» данных, состоящих из процедурных команд и инструкций SQL. В то время еще нельзя было строить модульные приложения или сохранять бизнес-правила на сервере. Технология SQL*Forms 3.0 позволяла создавать процедуры и функции на стороне клиента, хотя поддержка этих функций еще не была документирована, и поэтому многие разработчики ими не пользовались. Кроме того, в этой версии PL/SQL не поддерживались массивы и отсутствовало взаимодействие с операционной системой (для ввода и вывода). Так что до полноценного языка программирования было еще очень далеко.

Однако, несмотря на все ограничения, PL/SQL был очень тепло и даже с энтузиазмом принят сообществом разработчиков. Уже очень остро стала ощущаться потребность хотя бы в таких элементарных средствах, как условная команда `IF` в SQL*Forms. Необходимость в пакетном выполнении последовательности команд SQL стала безотлагательной.

В то время лишь немногие разработчики понимали, что исходная мотивация и побудительные причины для развития PL/SQL выходили за пределы потребности программного управления в таких продуктах, как SQL *Forms. Еще на ранней стадии жизненного цикла базы данных Oracle и ее инструментария компания Oracle Corporation выявила две ключевые слабости своей архитектуры: плохую переносимость и проблемы с полномочиями выполнения.

Улучшение переносимости приложений

Тем, кто знаком с маркетинговой и технической стратегией Oracle Corporation, может показаться странным сама постановка вопроса. Ведь с начала 1980-х годов именно переносимость была одной из сильных сторон решений Oracle. На момент выхода PL/SQL реляционные базы данных на основе языка С работали во многих операционных системах и на множестве аппаратных платформ. SQL*Plus и SQL*Forms легко адаптировались к разнообразным терминальным конфигурациям. Однако для решения многих задач по-прежнему требовались более совершенные и точные средства управления таких языков, как COBOL, С и FORTRAN. И стоило разработчику выйти за рамки платформенно-нейтрального инструментария Oracle, приложение утрачивало переносимость.

Язык PL/SQL расширяет диапазон требований к приложениям, которые полностью реализуются программными средствами, независимыми от операционной системы. В настоящее время для создания таких универсальных приложений применяется Java и другие языки программирования. Тем не менее PL/SQL занимает особое место как один из пионеров в этой области и, конечно, продолжает применяться разработчиками для создания легко переносимого кода приложений баз данных.

Улучшенная защита приложений и защита целостности транзакций

Защита приложений была еще более важным аспектом, чем переносимость. База данных и язык SQL позволяют жестко контролировать доступ к любым таблицам базы данных (и внесение изменений в них). Например, с командой **GRANT** вы можете быть уверены в том, что выполнение команды **UPDATE** с конкретной таблицей будет разрешено только определенным ролям и пользователям. С другой стороны, команда **GRANT** не может гарантировать, что пользователь внесет правильную последовательность изменений в одну или несколько таблиц, необходимых для большинства бизнес-транзакций.

Язык PL/SQL обеспечивает строгий контроль за выполнением логических транзакций. Одним из средств такого контроля является система полномочий на выполнение. Вместо того чтобы выдавать разрешения на обновление таблиц ролям или пользователям, вы выдаете разрешение только на выполнение процедуры, которая управляет и предоставляет доступ к необходимым структурам данных. Владельцем процедуры является другая схема базы данных Oracle («определитель»), которой, в свою очередь, предоставляются разрешения на обновление таблиц, участвующих в транзакции. Таким образом, процедура становится «привратником» транзакции. Программный код (будь то приложение Oracle Forms или исполняемый файл Pro*C) может выполняться только посредством вызова процедуры, и это гарантирует целостность транзакций приложения.

Начиная с Oracle8i, в Oracle появилось ключевое слово **AUTHID**, значительно расширяющее гибкость модели защиты PL/SQL. Оно позволяет выполнять программы в соответствии с описанной ранее моделью прав определителя или же выбрать режим **AUTHID CURRENT_USER**, при котором программы выполняются с правами текущей учетной записи. Это всего лишь один из примеров развития и повышения гибкости PL/SQL.

Скромное начало, постоянное усовершенствование

Язык SQL, каким бы мощным он ни был, не обладает гибкостью и мощностью, необходимой для создания полноценных приложений. В то же время PL/SQL, оставаясь в пределах не зависящего от операционной системы окружения Oracle, позволяет разрабатывать высокоэффективные приложения, удовлетворяющие потребностям пользователей.

Со времени своего появления PL/SQL прошел очень длинный путь. Разработчики, которые пользовались его первой версией, слишком уж часто вынуждены были говорить: «В PL/SQL это сделать невозможно». Сейчас это утверждение из факта постепенно превращается в отговорку. Если вы сталкиваетесь с задачей, которую, как вам кажется, нельзя решить средствами PL/SQL, не пытайтесь убедить в этом свое начальство. Копайте глубже, исследуйте возможности самого языка и пакетов PL/SQL, и необходимые средства, скорее всего, найдутся.

За прошедшие годы компания Oracle Corporation продемонстрировала свою приверженность PL/SQL. С выходом каждой новой версии базы данных Oracle в PL/SQL вводились все новые фундаментальные усовершенствования. Было разработано множество внешних (и встроенных) пакетов, расширяющих функциональность PL/SQL. В частности, язык был дополнен объектно-ориентированными возможностями, разнообразными структурами данных, усовершенствованный компилятор оптимизировал код и выдавал предупреждения о возможных проблемах с качеством и быстродействием кода. В целом язык был заметно углублен и расширен.

В следующем разделе представлены примеры программ PL/SQL, которые познакомят читателя с основами программирования PL/SQL.

Итак, PL/SQL

Если вы только приступаете к изучению программирования или еще не освоили ни PL/SQL, ни даже SQL, может показаться, что перед вами очень сложная задача. Но это не так. Она наверняка окажется намного проще, чем вы думаете.

Для оптимизма есть два основания:

- Выучить компьютерный язык значительно проще, чем второй или третий «человеческий» язык. Почему? Да потому, что компьютеры не особенно умны (они «думают» — выполняют операции — быстро, но отнюдь не творчески). Чтобы объяснить компьютеру, что он должен делать, приходится пользоваться очень жестким синтаксисом. Следовательно, язык, на котором мы с ним общаемся, тоже должен быть очень жестким (никаких исключений!) и поэтому выучить его несложно.
- Язык PL/SQL прост в сравнении с другими языками программирования. Программа делится на «блоки» с разными разделами, четко идентифицируемыми с помощью понятных ключевых слов.

Рассмотрим несколько примеров, демонстрирующих применение ключевых элементов структуры и функциональности PL/SQL.

Интеграция с SQL

Одним из важнейших аспектов PL/SQL является его тесная интеграция с SQL. Для выполнения SQL-инструкций в программах на PL/SQL не требуется никакой промежуточной программной «прослойки» вроде ODBC (Open Database Connectivity) или JDBC (Java Database Connectivity). Инструкция UPDATE или SELECT просто вставляется в программный код, как в следующем примере.

```
1 DECLARE
2     l_book_count INTEGER;
3
4 BEGIN
5     SELECT COUNT(*)
6         INTO l_book_count
7         FROM books
```

```

8      WHERE author LIKE '%FEUERSTEIN, STEVEN%';
9
10     DBMS_OUTPUT.PUT_LINE (
11       'Стивен является автором (или соавтором) ' ||
12       l_book_count ||
13       ' книг. ');
14
15     -- Я решил изменить написание своего имени ...
16     UPDATE books
17       SET author = REPLACE (author, 'STEVEN', 'STEPHEN')
18     WHERE author LIKE '%FEUERSTEIN, STEVEN%';
19 END;
```

Теперь посмотрим, что делает этот код. Его подробное описание дано в следующей таблице.

| Строки | Описание |
|--------|--|
| 1–3 | Объявление так называемого анонимного блока PL/SQL, в котором объявляется целочисленная переменная для хранения данных о количестве книг, автором или соавтором является Стивен Фейерштейн. (Подробнее о структуре блока в PL/SQL рассказывается в главе 3) |
| 4 | Ключевое слово BEGIN указывает на начало исполняемого раздела — кода, который будет выполнен при передаче этого блока в SQL*Plus |
| 5–8 | Запрос, определяющий общее количество книг, автором или соавтором которых является Стивен Фейерштейн. Особенно интересна строка 6: использованная в ней секция INTO на самом деле не является частью инструкции SQL, а связывает базу данных с локальными переменными PL/SQL |
| 10–13 | Для вывода количества книг используется встроенная процедура DBMS_OUTPUT.PUT_LINE (то есть процедура из пакета DBMS_OUTPUT, входящего в состав Oracle) |
| 15 | Однострочный комментарий, объясняющий назначение инструкции UPDATE |
| 16–18 | Чтобы изменить написание имени автора на Stephen, необходимо обновить таблицу books. Поиск всех вхождений слова STEVEN и замена их на STEPHEN осуществляется встроенной функцией REPLACE |

Управляющие конструкции и логические условия

PL/SQL содержит полный набор команд, предназначенных для управления последовательностью выполнения строк программы. В него входят следующие команды:

IF и CASE. Реализация условной логики выполнения — например, «Если количество книг больше 1000, то...»

Полный набор команд циклов и итеративных вычислений. К этой группе относятся команды FOR, WHILE и LOOP.

GOTO. Да, в PL/SQL есть даже GOTO — команда безусловной передачи управления из одной точки программы в другую. Впрочем, это не означает, что ей следует *пользоваться*.

Следующая процедура (многократно используемый блок кода, который можно вызывать по имени) демонстрирует работу отдельных команд:

```

1  PROCEDURE pay_out_balance (
2    account_id_in IN accounts.id%TYPE)
3  IS
4    l_balance_remaining NUMBER;
5  BEGIN
6    LOOP
7      l_balance_remaining := account_balance (account_id_in);
8
9      IF l_balance_remaining < 1000
10     THEN
11       EXIT;
12     ELSE
```

```

13         apply_balance (account_id_in, l_balance_remaining);
14     END IF;
15 END LOOP;
16 END pay_out_balance;

```

Структура программного кода описана в следующей таблице.

| Строки | Описание |
|--------|---|
| 1–2 | Заголовок процедуры, уменьшающей баланс банковского счета с целью оплаты счетов. В строке 2 перечислен список параметров процедуры, состоящий из одного входного значения (идентификационного номера банковского счета) |
| 3–4 | Раздел объявлений процедуры. Обратите внимание: вместо ключевого слова DECLARE, как в предыдущем примере, я использую ключевое слово IS (или AS) для отделения заголовка от объявлений |
| 6–15 | Пример простого цикла LOOP. Команда EXIT (строка 11) определяет условие завершения цикла; в циклах FOR и WHILE условие завершения цикла определяется по-другому |
| 7 | Вызов функции account_balance, определяющей баланс счета. Это пример вызова одной многократно используемой программы из другой. Вызов процедуры из другой продемонстрирован в строке 13 |
| 9–14 | Команда IF, которую можно интерпретировать так: «Если баланс счета окажется меньше 1000 долларов, прекратить оплату счетов. В противном случае оплатить следующий счет» |

Обработка ошибок

Язык PL/SQL предоставляет разработчикам мощный механизм оповещения о возникающих ошибках и их обработки. Следующая процедура получает имя и баланс счета по идентификатору, после чего проверяет баланс. При слишком низком значении процедура явно инициирует исключение, которое прекращает выполнение программы:

```

1  PROCEDURE check_account (
2      account_id_in IN accounts.id%TYPE)
3  IS
4      l_balance_remaining      NUMBER;
5      l_balance_below_minimum  EXCEPTION;
6      l_account_name           accounts.name%TYPE;
7  BEGIN
8      SELECT name
9          INTO l_account_name
10         FROM accounts
11        WHERE id = account_id_in;
12
13      l_balance_remaining := account_balance (account_id_in);
14
15      DBMS_OUTPUT.PUT_LINE (
16          'Баланс счета ' || l_account_name ||
17          ' = ' || l_balance_remaining);
18
19      IF l_balance_remaining < 1000
20      THEN
21          RAISE l_balance_below_minimum;
22      END IF;
23
24  EXCEPTION
25      WHEN NO_DATA_FOUND
26      THEN
27          -- Ошибочный идентификатор счета
28          log_error (...);
29          RAISE;
30      WHEN l_balance_below_minimum
31      THEN
32          log_error (...);
33          RAISE VALUE_ERROR;
34  END;

```

Рассмотрим подробнее ту часть кода, которая связана с обработкой ошибок.

| Строки | Описание |
|--------|---|
| 5 | Объявление пользовательского исключения с именем <code>I_balance_below_minimum</code> . В Oracle имеется набор заранее определенных исключений, таких как <code>DUP_VAL_ON_INDEX</code> , но для данного приложения я хочу создать нечто более конкретное, поэтому определяю собственный тип исключения |
| 8–11 | Запрос для получения имени счета. Если счет с указанным идентификатором не существует, Oracle инициирует стандартное исключение <code>NO_DATA_FOUND</code> , что ведет к завершению программы |
| 19–22 | Если баланс слишком низок, процедура явно инициирует пользовательское исключение, поскольку это свидетельствует о наличии серьезных проблем со счетом |
| 24 | Ключевое слово <code>EXCEPTION</code> отмечает конец исполняемого раздела и начало раздела исключений, в котором обрабатываются ошибки |
| 25–28 | Блок обработки ошибок для ситуации, когда счет не найден. Если было инициировано исключение <code>NO_DATA_FOUND</code> , здесь оно перехватывается, а ошибка регистрируется в журнале процедурой <code>log_error</code> . Затем я заново инициирую <i>то же самое исключение</i> , чтобы внешний блок был в курсе того, что для идентификатора счета отсутствует совпадение |
| 30–33 | Блок обработки ошибок для ситуации, когда баланс счета оказался слишком низким (пользовательское исключение для данного приложения). Если было инициировано исключение <code>I_balance_below_minimum</code> , оно перехватывается и ошибка регистрируется в журнале. Затем я инициирую системное исключение <code>VALUE_ERROR</code> , чтобы оповестить внешний блок о проблеме |

Механизмы обработки ошибок PL/SQL подробно рассматриваются в главе 6.

Конечно, о PL/SQL еще можно сказать очень много — собственно, именно поэтому материал этой книги занимает не одну сотню страниц! Но эти примеры дают некоторое представление о коде PL/SQL, его важнейших синтаксических элементах и о той простоте, с которой пишется (и читается) код PL/SQL.

О версиях

Каждая версия базы данных Oracle выходит с собственной версией PL/SQL, расширяющей функциональные возможности языка. Поэтому каждый программист должен прикладывать немало усилий, чтобы просто не отставать от эволюции PL/SQL. Необходимо постоянно осваивать новые возможности каждой версии, чтобы знать, как пользоваться ими в приложениях, и решать, оправдана ли их полезность модификацию уже существующих приложений.

В табл. 1.1 представлены основные средства всех версий PL/SQL — как старых, так и современных. (Учтите, что в ранних версиях Oracle номера версий PL/SQL отличались от версий базы данных, но начиная с Oracle8, версии совпадают.) В таблице приведен очень краткий обзор новых возможностей каждой версии. После таблицы следуют более подробные описания новых возможностей PL/SQL новейшей версии Oracle — Oracle Database 12c.



Каждый пакет Oracle Developer содержит собственную версию PL/SQL, которая обычно отстает от версии, доступной в самой СУБД. В этой главе (как и во всей книге) основное внимание уделяется программированию PL/SQL на стороне сервера.

Таблица 1.1. Версии Oracle и PL/SQL

| Версия Oracle | Характеристики |
|---------------|---|
| 6.0 | Исходная версия PL/SQL (1.0), использовавшаяся главным образом как сценарный язык в SQL*Plus (она еще не позволяла создавать именованные программы с возможностью многократного вызова) и как язык программирования в SQL*Forms3 |
| 7.0 | Обновление первой версии (2.0). Добавлена поддержка хранимых процедур, функций, пакетов, определяемых программистом записей, таблиц PL/SQL и многочисленных расширений |
| 7.1 | Версия PL/SQL (2.1) поддерживает определяемые программистом подтипы данных, возможность использования хранимых функций в SQL-инструкциях, динамический SQL (посредством пакета DBMS_SQL). С появлением версии PL/SQL 2.1 стало возможным выполнять DDL из программ PL/SQL |
| 7.3 | В этой версии PL/SQL (2.3) были расширены возможности коллекций, усовершенствовано удаленное управление связями между таблицами, добавлены средства файлового ввода/вывода (пакет UTF_FILE) и завершена реализация курсорных переменных |
| 8.0 | Новый номер версии (8.0) отражает стремление Oracle синхронизировать номера версий PL/SQL с соответствующими номерами версий СУБД. PL/SQL 8.0 поддерживает многие усовершенствования Oracle8, включая большие объекты (LOB), объектно-ориентированную структуру и принципы разработки, коллекции (VARRAY и вложенные таблицы), а также средство организации очередей Oracle/AQ (Oracle/Advanced Queuing) |
| 8.1 | Это первая из i-серий Oracle (базы данных для Интернета). В соответствующую версию PL/SQL включен впечатляющий набор новых средств и возможностей, в том числе новая версия динамического SQL, поддержка Java для доступа к базе данных, модель процедур с правами вызывающего, разрешения на выполнение, автономные транзакции, высокопроизводительный «пакетный» язык DML и запросы |
| 9.1 | Версия Oracle 9i Database Release 1 появилась вскоре после Oracle 8i. В ее первом выпуске была реализована поддержка наследования объектных типов, табличные функции и выражения с курсорами (позволяющие организовать параллельное выполнение функций PL/SQL), многоуровневые коллекции, конструкция CASE и выражения CASE |
| 9.2 | В Oracle 9i Database Release 2 главный акцент сделан на языке XML, но есть и другие усовершенствования для разработчиков PL/SQL: ассоциативные массивы (индексируемые не только целыми числами, но и строками VARCHAR2), язык DML на базе записей (позволяющий, например, выполнять вставку с использованием записи), а также многочисленные улучшения пакета UTL_FILE (который теперь позволяет выполнять чтение/запись файлов из программы PL/SQL) |
| 10.1 | Версия Oracle Database 10g Release 1 была выпущена в 2004 году. Основное внимание в ней уделялось решетчатой обработке данных, с улучшенным/автоматизированным управлением базой данных. С точки зрения PL/SQL, самые важные новые функции, оптимизированный компилятор и предупреждения на стадии компиляции, были введены абсолютно прозрачно для разработчиков |
| 10.2 | Версия Oracle Database 10g Release 2, выпущенная в 2005 году, содержала небольшое количество новых функций для разработчиков PL/SQL — прежде всего, пре-процессор с возможностью условной компиляции фрагментов программного кода в зависимости от определяемых разработчиком логических условий |
| 11.1 | Версия Oracle Database 11g Release 1 появилась в 2007 году. Важнейшей новой функцией для разработчиков PL/SQL был кэш результатов функций, но появились и другие удобства: составные триггеры, команда CONTINUE и низкоуровневая компиляция, генерирующая машинный код |
| 11.2 | Версия Oracle Database 11g Release 2 вышла осенью 2009 года. Самым важным новшеством стала возможность оперативного переопределения, позволяющая администраторам изменять приложения на ходу, во время их выполнения пользователями |
| 12.1 | Версия Oracle Database 12c Release 1 вышла в июне 2013 года. Она предлагает ряд усовершенствований в области управления доступом и привилегиями программных модулей и представлений; обеспечивает дополнительную синхронизацию языков SQL и P, особенно в отношении максимальной длины VARCHAR2 и динамического связывания SQL; поддерживает определение простых функций в конструкциях SQL и добавляет пакет UTL_CALL_STACK для детализированного управления доступом к стеку вызовов, стеку ошибок и обратной трассировке |

Новые возможности PL/SQL в Oracle Database 12c

Oracle Database 12c предоставляет ряд новых возможностей, которые повышают производительность кода PL/SQL и делают его более удобным. Также были устранены некоторые недоработки в языке. Ниже приводится сводка важнейших изменений языка для разработчиков PL/SQL.

Проникновение новых типов данных PL/SQL через границу PL/SQL/SQL

До выхода версии 12.1 привязка типов данных, специфических для PL/SQL (например, ассоциативных массивов), в динамических командах SQL была невозможна. Теперь значения типов данных, поддерживаемых только в PL/SQL, могут привязываться в анонимных блоках, вызовах функций PL/SQL в запросах SQL, командах CALL и в операторе TABLE в запросах SQL.

Условие ACCESSIBLE_BY

Теперь в спецификацию пакета можно добавить секцию ACCESSIBLE_BY, указывающую, какие программные модули могут вызывать подпрограммы этого пакета. Данная возможность позволяет открыть доступ к подпрограммам во вспомогательных пакетах, предназначенных только для конкретных программных модулей. Фактически речь идет о своего рода «белых списках» для пакетов.

Неявные результаты команд

До выхода Oracle Database 12c хранимая подпрограмма PL/SQL явно возвращала итоговые наборы из запросов SQL, через параметр OUT REF CURSOR или условие RETURN. Клиентская программа должна была осуществить явную привязку параметров для получения итогового набора. Теперь хранимая подпрограмма PL/SQL может неявно вернуть результаты запроса клиенту при помощи пакета PL/SQL DBMS_SQL (вместо параметров OUT REF CURSOR). Эта функциональность упрощает миграцию приложений, полагающихся на неявное возвращение результатов запросов из хранимых подпрограмм (возможность поддерживается такими языками, как Transact SQL) из сторонних баз данных в Oracle.

Представления BEQUEATH CURRENT_USER

До выхода Oracle Database 12c представление всегда вело себя так, как если бы оно обладало правами определяющего (AUTHID DEFINER), даже если оно использовалось в модуле, обладающем правами вызывающего (AUTHID CURRENT_USER). Теперь представление может определяться с ключевыми словами BEQUEATH DEFINER (используется по умолчанию), при котором оно использует поведение с правами определяющего, или BEQUEATH CURRENT_USER, с которыми его поведение близко к поведению с правами вызывающего.

Предоставление ролей программным модулям

До выхода Oracle Database 12c модули с правами вызывающего всегда выполнялись с привилегиями вызывающей стороны. Если вызывающая сторона имела более высокие привилегии, чем владелец, то модуль с правами вызывающего мог выполнять операции, непредусмотренные владельцем (или запрещенные для него).

В версии 12.1 вы можете назначать роли отдельным пакетам PL/SQL и автономным подпрограммам. Вместо модуля с правами определяющего можно создать модуль с правами вызывающего и назначить ему нужные роли. В этом случае модуль с правами вызывающего выполняется с привилегиями вызывающего и ролями, но без дополнительных привилегий, присущих схеме определяющего.

Модуль с правами вызывающего теперь может выполняться с привилегиями вызывающего только в том случае, если его владелец обладает привилегией `INHERIT PRIVILEGES` для вызывающего или привилегией `INHERIT ANY PRIVILEGES`.



Привилегия `INHERIT PRIVILEGES` предоставляется всем схемам при установке/обновлении.

Новые директивы условной компиляции

В версии 12.1 в Oracle были добавлены две новые стандартные директивы `$$PLSQL_UNIT_OWNER` и `$$PLSQL_UNIT_TYPE`, которые возвращают владельца и тип текущего программного модуля PL/SQL.

Оптимизация выполнения функций в SQL

Oracle предоставляет два новых способа улучшения производительности функций PL/SQL в командах SQL: разработчик может определить функцию внутри команды SQL при помощи условия `WITH` или добавить в программный модуль директиву UDF, которая сообщает компилятору, что функция будет использоваться в основном в командах SQL.

Использование `%ROWTYPE` с невидимыми столбцами

Oracle Database 12c позволяет определять невидимые столбцы. В PL/SQL атрибут `%ROWTYPE` поддерживает такие столбцы и работу с ними.

FETCH FIRST и BULK COLLECT

В версии 12.1 можно использовать необязательное условие `FETCH FIRST` для ограничения количества строк, возвращаемых запросом, что существенно снижает SQL-сложность стандартных запросов «верхних N результатов». Наибольшую пользу `FETCH FIRST` принесет при миграции сторонних баз данных в Oracle Database. Впрочем, эта конструкция также способна повысить быстродействие некоторых команд `SELECT BULK COLLECT INTO`.

Пакет `UTL_CALL_STACK`

До выхода версии 12.1 пакет `DBMS_UTILITY` предоставлял три функции (`FORMAT_CALL_STACK`, `FORMAT_ERROR_STACK` и `FORMAT_ERROR_BACKTRACE`) для получения информации о стеке вызовов, стеке ошибок и обратной трассировке соответственно. В версии 12.1 пакет `UTL_CALL_STACK` предоставляет ту же информацию, а также более точные средства управления доступом к содержимому этих отформатированных строк.

Ресурсы для разработчиков PL/SQL

Первое издание данной книги вышло в 1995 году. Тогда это событие произвело настоящую сенсацию — это была первая независимая (то есть не связанная с компанией Oracle) книга о PL/SQL, которую давно и с нетерпением ожидали разработчики во всем мире. С тех пор появилось множество PL/SQL-ресурсов, среди которых различного рода книги, среды разработки, утилиты и веб-сайты (но разумеется, эта книга остается самым важным и ценным из них!).

Многие из этих ресурсов кратко описаны в следующих разделах. Пользуясь этими ресурсами, многие из которых относительно недороги, а часть вообще предоставляется бесплатно, вы сможете значительно повысить свою квалификацию (и качество вашего кода).

Книги о PL/SQL от O'Reilly

За прошедшие годы серия книг о PL/SQL издательства O'Reilly & Associates представлена уже довольно внушительным списком. Более подробная информация об этих изданиях представлена на сайте издательства.

Oracle PL/SQL Programming (авторы Steven Feuerstein, Bill Pribyl). Книга в 1300 страниц, которую вы сейчас читаете. Это настольная книга для многих профессиональных программистов PL/SQL, в которой рассматриваются все возможности базового языка. В этой версии описана версия Oracle11g Release 2.

Learning Oracle PL/SQL (авторы Bill Pribyl, Steven Feuerstein). Упрощенное введение в язык PL/SQL для новичков в программировании и тех, кто переходит к PL/SQL с других языков.

Oracle PL/SQL Best Practices (автор Steven Feuerstein). Относительно небольшая книга с десятками полезных советов, рекомендаций и приемов, которые помогут читателю писать качественный код PL/SQL. Эта книга является чем-то вроде краткого конспекта по PL/SQL. Второе издание содержит полностью переработанный материал, который строится на истории группы разработчиков из вымышленной компании My Flimsy Excuse.

Oracle PL/SQL Developer's Workbook (авторы Steven Feuerstein, Andrew Odewahn). Содержит серию вопросов и ответов, помогающих разработчику проверить и дополнить свои знания о языке. В книге рассматриваются возможности языка вплоть до версии Oracle8i, но, конечно, большинство примеров работает и в последующих версиях базы данных.

Oracle Built-in Packages (авторы Steven Feuerstein, Charles Dye, John Beresniewicz). Справочное руководство по всем стандартным пакетам, входящим в комплект поставки Oracle. Эти пакеты позволяют упростить трудную работу, а иногда даже сделать невозможное. Рассматриваются версии до Oracle8 включительно, но подробные объяснения и примеры будут чрезвычайно полезны и в последующих версиях базы данных.

Oracle PL/SQL for DBAs (авторы Arup Nanda, Steven Feuerstein). С выходом каждой новой версии Oracle язык PL/SQL играет все более важную роль в работе администраторов баз данных (БД). Это объясняется двумя основными причинами. Во-первых, многие административные функции доступны через интерфейс пакетов PL/SQL. Чтобы пользоваться ими, необходимо также писать и запускать программы PL/SQL. Во-вторых, практический опыт PL/SQL необходим администратору БД и для того, чтобы выявлять проблемы в чужом коде. Материал, представленный в книге, поможет администратору БД быстро освоить PL/SQL для повседневной работы.

Oracle PL/SQL Language Pocket Reference (Steven Feuerstein, Bill Pribyl, Chip Dawes). Маленький, но очень полезный справочник с описанием базовых элементов языка PL/SQL вплоть до Oracle11g.

Oracle PL/SQL Built-ins Pocket Reference (авторы Steven Feuerstein, John Beresniewicz, Chip Dawes). Еще один лаконичный справочник с краткими описаниями всех функций и пакетов вплоть до Oracle8.

PL/SQL в Интернете

Программисты PL/SQL найдут в Сети много полезных ресурсов. В списке в основном представлены те ресурсы, для которых соавторы предоставляли свои материалы или помогали управлять контентом.

PL/SQL Obsession. Интернет-портал Стивена Фейерштейна содержит ссылки на различные ресурсы PL/SQL: презентации, примеры кода, бесплатные программы (некоторые из них упоминаются в следующем разделе), видеоролики и т. д.

PL/SQL Challenge. Сайт, основанный на концепции «активного изучения», — вместо того, чтобы читать книгу или веб-страницу, вы отвечаете на вопросы по PL/SQL, SQL, логике, проектированию баз данных и Oracle Application Express, проверяя свои познания в этих областях.

PL/SQL Channel. Библиотека видеороликов (на 27 с лишним часов) по языку Oracle PL/SQL, записанных Стивеном Фейерштейном.

Oracle Technology Network. Сайт OTN (Oracle Technology Network) «предоставляет сервисы и ресурсы, необходимые разработчикам для создания, тестирования и распространения приложений» на основе технологии Oracle. Он знаком миллионам разработчиков: здесь можно загрузить программное обеспечение, документацию и множество примеров кода. PL/SQL также имеет собственную страницу на сайте OTN.

Quest Error Manager. Инфраструктура для стандартизации обработки ошибок в приложениях на базе PL/SQL. При помощи QEM вы сможете организовать регистрацию и оповещение об ошибках через универсальный API с минимальными усилиями. Информация об ошибках сохраняется в таблицах экземпляров (общая информация об ошибке) и контекста (пары «имя-значение», смысл которых определяется конкретным приложением).

oracle-developer.net. Сайт поддерживается Эдрианом Биллингтоном (написавшим раздел о конвейерных функциях в главе 21). Он предоставляет разработчикам баз данных Oracle замечательную подборку статей, учебников и вспомогательных инструментов. Эдриан углубленно рассматривает новые возможности каждой версии Oracle Database, приводя многочисленные примеры, сценарии анализа производительности и т. д.

ORACLE-BASE. ORACLE-BASE — еще один превосходный ресурс для специалистов по технологиям Oracle, созданный и сопровождаемый экспертом по Oracle: Тимом Холлом. Тим является обладателем звания Oracle ACE Director, участником сети OakTable Network, а также обладателем премии Oracle Magazine Editor's Choice Awards в номинации «ACE of the Year». Занимается администрированием, проектированием и разработкой баз данных Oracle с 1994 года. См. <http://oracle-base.com>.

Несколько советов

С 1995 года, когда вышло в свет первое издание настоящей книги, мне представилась возможность обучать десятки тысяч разработчиков PL/SQL, помогать им и сотрудничать с ними. За это время мы многому научились и у наших читателей, составили более полное представление об оптимальных методах работы с PL/SQL. Надеюсь, вы позволите нам поделиться с вами нашими представлениями о том, как эффективнее работать с таким мощным языком программирования.

Не торопитесь!

Мы почти всегда работаем в очень жестких временных рамках. Времени вечно не хватает, ведь нам за короткое время нужно написать огромное количество кода. Итак, нужно поскорее приступить к работе — не так ли?

Нет, не так. Если сразу же углубиться в написание программного кода, бездумно преобразуя постановку задачи в сотни, тысячи и даже десятки тысяч строк, получится просто «каша», которую не удастся ни отладить, ни снабдить достойным сопровождением. Но и в жесткий график вполне можно уложиться, если не поддаваться панике и тщательно все спланировать.

Мы настоятельно рекомендуем не поддаваться давлению времени. Тщательно подготовьтесь к написанию нового приложения или программы.

1. *Создайте сценарии тестирования.* До того как будет написана первая строка кода, необходимо продумать, каким образом будет проверяться реализация задачи. Это позволит вам заранее продумать интерфейс программы и ее функциональность.
2. *Установите четкие правила написания SQL-команд в приложении.* В общем случае мы рекомендуем разработчикам не включать в программы большой объем SQL-кода. Большинство инструкций, в том числе запросы на обновление и вставку отдельных записей, должны быть «скрыты» в заранее написанных и тщательно отлаженных процедурах и функциях (это называется *инкапсуляцией данных*). Такие программы оптимизируются, тестируются и сопровождаются более эффективно, чем SQL-инструкции, хаотично разбросанные по программному коду (многие из которых неоднократно повторяются в приложении).
3. *Установите четкие правила обработки исключений в приложении.* Желательно создать единый пакет обработки ошибок, который скрывает все подробности ведения журнала ошибок, определяет механизм инициирования исключений и их распространения во вложенных блоках, а также позволяет избежать жесткого кодирования исключений, специфических для конкретного приложения, прямо в программном коде. Этим пакетом должны пользоваться все разработчики — в таком случае им не придется писать сложный, отнимающий много времени код обработки ошибок.
4. *Выполняйте «пошаговую проработку».* Другими словами, придерживайтесь принципа нисходящего проектирования, чтобы уменьшить сложность кода, с которым вы имеете дело в каждый конкретный момент. Применяя данный подход, вы обнаружите, что исполняемые разделы ваших модулей становятся короче и проще для понимания. В будущем такой код будет проще сопровождать и модифицировать. Важнейшую роль в реализации этого принципа играет использование локальных, или вложенных, модулей.

Это лишь некоторые важные аспекты, на которые следует обратить внимание, приступая к написанию программного кода. Помните, что спешка при разработке приводит к многочисленным ошибкам и огромным потерям времени.

Не бойтесь обращаться за помощью

Вы, профессиональный программист, наверняка очень умны, много учились, повышали свою квалификацию и теперь неплохо зарабатываете. Вам под силу решить практически любую задачу и вы этим гордитесь. Но к сожалению, успехи нередко делают нас самоуверенными, и мы не любим обращаться за помощью даже при возникновении серьезных затруднений. Такое отношение к делу опасно и деструктивно.

Программное обеспечение пишется людьми. И поэтому очень важно понимать, что огромную роль в его разработке играет психологический фактор. Приведу простой пример.

У Джо, руководителя группы из шести разработчиков, возникает проблема с его кодом. Он бьется над ней часами, все больше выходя из себя, но так и не может найти источник ошибки. Ему и в голову не приходит обратиться за помощью к коллегам, потому что он является самым опытным во всей группе. Наконец, Джо доходит «до точки» и сдается. Он со вздохом снимает телефонную трубку и набирает добавочный: «Сандра, ты можешь зайти и взглянуть на мою программу? У меня какая-то проблема, с которой я никак не могу разобраться». Сандра заходит и с первого взгляда на программу Джо указывает на то, что должно было быть очевидно несколько часов назад. Ура! Программа исправлена, Джо благодарит, но на самом деле он тайно переживает.

У него в голове проносятся мысли «Почему же я этого не заметил?» и «А если бы я потратил еще пять минут на отладку, то нашел бы сам». Все это понятно, но довольно глупо. Короче говоря, часто мы не можем найти свои проблемы, потому что находимся слишком близко к собственному коду. Иногда нужен просто свежий взгляд со стороны, и это не имеет ничего общего ни с опытом, ни с особыми знаниями.

Именно поэтому мы рекомендуем придерживаться следующих простых правил.

- *Не бойтесь признаться в том, что чего-то не знаете.* Если вы будете скрывать, что вам не все известно о приложении или его программном коде, это кончится плохо. Выработайте в коллективе культуру взаимоотношений, при которой вполне естественными считаются вопросы друг к другу и обоюдные консультации.
- *Обращайтесь за помощью.* Если вы не можете найти причину ошибки в течение получаса, немедленно просите о помощи. Можно даже организовать систему взаимопомощи, при которой у каждого работника имеется напарник. Не допускайте, чтобы вы (или ваши коллеги) часами мучились с поисками выхода из затруднительного положения.
- *Организируйте процесс рецензирования кода.* Код должен передаваться в отдел контроля качества или использоваться в проекте только после его предварительного прочтения и конструктивной критики одним или несколькими разработчиками.

Поощряйте творческий (и даже радикальный) подход к разработке

Мы склонны превращать в рутину практически все составляющие нашей жизни. Мы привыкаем писать код определенным образом, делаем определенные допущения о продукте, отвергаем возможные решения без серьезного анализа, потому что заранее знаем, что это сделать нельзя. Разработчики крайне необъективны в оценке своих программ и часто закрывают глаза на явные недостатки. Иногда можно услышать: «Этот код не будет работать быстрее», «Я не могу сделать то, что хочет пользователь; придется подождать следующей версии», «С продуктом X, Y или Z все было бы легко и быстро, а с такими средствами приходится буквально сражаться за каждую мелочь».

Но на самом деле выполнение практически любого кода можно немного ускорить. И программа может работать *именно так*, как хочет пользователь. И хотя у каждого продукта имеются свои ограничения, сильные и слабые стороны, не нужно дожидаться выхода следующей версии. Лучше встретить проблему лицом к лицу и, не позволяя себе никаких отговорок, найти ее решение.

Как это сделать? Следует отказаться от некоторых представлений и посмотреть на мир свежим взглядом. Пересмотрите выработанные годами профессиональные привычки. Относитесь к задаче творчески — постарайтесь отступить от традиционных методов, от зачастую ограниченных и механистических подходов.

Не бойтесь экспериментировать, даже если ваши идеи покажутся радикальным отступлением от нормы. Вы будете удивлены тем, как многому можно научиться таким образом; вырастаете как программист, способный к решению нестандартных задач. Многое становится возможным, когда вы перестаете говорить: «Это невозможно!», а, наоборот, спокойно киваете и бормочете: «А если попробовать так...»

2

Написание и запуск кода PL/SQL

Каждый программист PL/SQL — даже если он понятия не имеет о таких задачах, как системное проектирование или модульное тестирование, — должен владеть некоторыми основными навыками:

- Перемещение по базе данных.
- Создание и редактирование исходного кода PL/SQL.
- Компиляция исходного кода PL/SQL, исправление ошибок и предупреждений компилятора.
- Запуск откомпилированной программы из некоторой среды.
- Анализ результатов выполнения кода (экранный вывод, изменения в таблицах и т. д.).

В отличие от «самостоятельных» языков вроде С, язык PL/SQL работает под управлением операционной среды Oracle (то есть является *«встроенным»* языком), поэтому во всех перечисленных задачах встречаются некоторые неожиданные нюансы — как положительные, так и отрицательные. В этой главе вы научитесь выполнять эти операции на простейшем уровне (с использованием SQL*Plus). Глава завершается практическими примерами вызова кода PL/SQL из распространенных сред программирования PHP, С и т. д. За подробной информацией о компиляции и других нетривиальных задачах обращайтесь к главе 20.

Перемещение по базе данных

Любая программа PL/SQL пишется для работы с содержимым базы данных Oracle. Вполне естественно, что программист должен уметь «заглянуть за кулисы» базы данных Oracle, в которой будет выполняться его код. Вы должны знать, как просмотреть структуры данных (таблицы, столбцы, последовательности, пользовательские типы и т. д.) и сигнатуры всех существующих хранимых программ, которые будут вызываться в вашем коде. Также необходимо уметь получать информацию о реальном содержимом таблиц (столбцы, ограничения и т. д.).

Существует два разных подхода к перемещению по базе данных:

- Использование интегрированной среды разработки (IDE) — Toad, SQL Developer, PL/SQL Developer, SQL Navigator и т. д. Все интегрированные среды предоставляют визуальные средства просмотра, в которых для перемещения используется мышь.
- Запуск в среде командной строки (такой, как SQL*Plus) сценариев, запрашивающих информацию из словарей данных: ALL_OBJECTS, USER_OBJECTS и т. д. (см. далее в этой главе).

Я настоятельно рекомендую использовать графическую среду. Если вы достаточно давно работаете с Oracle, возможно, вы уже привыкли к сценариям, и работа с ними проходит достаточно эффективно. Но для большинства рядовых пользователей графический интерфейс намного удобнее, а работа с ним получается более производительной.

В главе 20 также приведены примеры использования словарей данных для работы с кодовой базой PL/SQL.

Создание и редактирование исходного кода

В наши дни программистам доступен очень широкий выбор редакторов кода, от простейших текстовых редакторов до самых экзотических средств разработки. И программисты принимают разные решения. Один из авторов этой книги, Стивен Фейерштейн, является сторонником IDE Toad. Он принадлежит к числу типичных пользователей, которые знают всего 10% всех функций и кнопок, но интенсивно используют их в своей работе. Билл Прибыл, напротив, предпочитает называть себя «чудаком, предпочитающим писать программы PL/SQL в простейшем текстовом редакторе. И если и пользоваться какими-то удобствами, то разве что автоматической расстановкой отступов в коде, а также цветовым выделением ключевых слов, комментариев, литералов и переменных».

Современные редакторы для программистов не ограничиваются расстановкой отступов и цветовым выделением ключевых слов; они также поддерживают графическую отладку, автоматическое завершение ключевых слов, просмотр списка подпрограмм и пакетов в процессе ввода имени, отображение параметров подпрограмм и выделение позиции, в которой компилятор обнаружил ошибку. Некоторые редакторы также поддерживают «гиперссылки» для быстрого просмотра объявлений переменных и подпрограмм. Впрочем, все эти функции характерны для большинства компилируемых языков.

Уникальность PL/SQL заключается в том, что исходный код хранимых программ перед компиляцией и выполнением должен быть загружен в базу данных. Для загрузки кода, хранящегося в базе, программист обычно должен обладать достаточным уровнем разрешений. Естественно, сразу же возникает ряд организационных вопросов:

- Как и где программист находит «оригинал» хранимой программы?
- Хранится ли код на диске или существует только в базе данных?
- Как и с какой частотой должно выполняться резервное копирование?
- Как организовать доступ к коду для нескольких разработчиков? А именно существует ли система контроля версий?

На эти вопросы следует ответить до начала разработки приложения — лучше всего приняв решение о том, какие программные инструменты должны делать все это за вас. Не существует единого набора инструментов или методологий, оптимального для всех групп разработки, хотя я всегда храню «оригинал» исходного кода в файлах — использовать реляционную СУБД в качестве хранилища программного кода *не рекомендуется*.

В следующем разделе я покажу, как при помощи SQL*Plus выполняются многие основные операции разработки PL/SQL. Эти же операции также можно выполнять и в IDE.

SQL*Plus

Предок всех клиентских интерфейсов Oracle — приложение SQL*Plus — представляет собой *интерпретатор* для SQL и PL/SQL, работающий в режиме командной строки. Таким образом, приложение принимает от пользователя инструкции для доступа к базе данных, передает их серверу Oracle и отображает результаты на экране.

При всей примитивности пользовательского интерфейса SQL*Plus является одним из самых удобных средств выполнения кода Oracle. Здесь нет замысловатых «примочек» и сложных меню, и лично мне это *нравится*. Когда я только начинал работать с Oracle (примерно в 1986 году), предшественник SQL*Plus гордо назывался *UFI* — User Friendly Interface (дружественный пользовательский интерфейс). И хотя в наши дни даже самая новая версия SQL*Plus вряд ли завоюет приз за дружественный интерфейс, она, по крайней мере, работает достаточно надежно.

В предыдущие годы компания Oracle предлагала версии приложения SQL*Plus с разными вариантами запуска:

- **Консольная программа.** Программа выполняется из оболочки или командной строки (окружения, которое иногда называют консолью)¹.
- **Программа с псевдографическим интерфейсом.** Эта разновидность SQL*Plus доступна только в Microsoft Windows. Я называю этот интерфейс «псевдографическим», потому что он практически не отличается от интерфейса консольной программы, хотя и использует растровые шрифты. Учтите, что Oracle собирается свернуть поддержку данного продукта, и после выхода Oracle8i он фактически не обновлялся.
- **Запуск через iSQL*Plus.** Программа выполняется из браузера машины, на которой работает HTTP-сервер Oracle и сервер iSQL*Plus.

Начиная с Oracle11g, Oracle поставляет только консольную версию программы (sqlplus.exe).

Главное окно консольной версии SQL*Plus показано на рис. 2.1.

```

C:\cygdrive\c\Documents and Settings\bill\My Documents\books\oppt
$ sqlplus bob/swordfish

SQL*Plus: Release 10.1.0.3.0 - Production on Wed May 18 10:04:26 2005

Copyright (c) 1982, 2004, Oracle. All rights reserved.

Connected to:
Personal Oracle Database 10g Release 10.1.0.3.0 - Production
with the Partitioning, OLAP and Data Mining options

BOB@bp10gr1 > set linesize 95 pagesize 1000
BOB@bp10gr1 > column author format a30 word_wrap
BOB@bp10gr1 > column title format a45 word_wrap
BOB@bp10gr1 > select isbn, author, title from books
  2 where lower(author) like '%feuerstein%' AND lower(author) like '%pribyl%';

-----
ISBN              AUTHOR                                TITLE
-----
0-596-00381-1    Feuerstein, Steven, with Bill    Oracle PL/SQL Programming, Third Ed.
                Pribyl
0-596-00180-0    Bill Pribyl with Steven          Learning Oracle PL/SQL
                Feuerstein
0-596-00680-2    Feuerstein, Steven, Bill        Oracle PL/SQL Language Pocket Reference,
                Pribyl, Chip Dawes              Third Ed.

BOB@bp10gr1 >
  
```

Рис. 2.1. Окно приложения SQL*Plus в консольном сеансе

Лично я предпочитаю консольную программу остальным по следующим причинам:

- она быстрее перерисовывает экран, что важно при выполнении запросов с большим объемом выходных данных;

¹ Oracle называет это «версией SQL*Plus с интерфейсом командной строки», но мы полагаем, что это определение не однозначно, поскольку интерфейс командной строки предоставляют два из трех способов реализации SQL*Plus.

- у нее более полный журнал команд, вводившихся ранее в командной строке (по крайней мере на платформе Microsoft Windows);
- в ней проще менять такие визуальные характеристики, как шрифт, цвет текста и размер буфера прокрутки;
- она доступна практически на любой машине, на которой установлен сервер или клиентские средства Oracle.

Запуск SQL*Plus

Чтобы запустить консольную версию SQL*Plus, достаточно ввести команду `sqlplus` в приглашении операционной системы OS>:

```
OS> sqlplus
```

Этот способ работает как в операционных системах на базе Unix, так и в операционных системах Microsoft. SQL*Plus отображает начальную заставку, а затем запрашивает имя пользователя и пароль:

```
SQL*Plus: Release 11.1.0.6.0 - Production on Fri Nov 7 10:28:26 2008
```

```
Copyright (c) 1982, 2007, Oracle. All rights reserved.
```

```
Enter user-name: bob
```

```
Enter password: swordfish
```

```
Connected to:
```

```
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - 64bit
```

```
SQL>
```

Если появится приглашение SQL>, значит, все было сделано правильно. (Пароль, в данном случае `swordfish`, на экране отображаться не будет.)

Имя пользователя и пароль также можно указать в командной строке запуска SQL*Plus:

```
OS> sqlplus bob/swordfish
```

Однако так поступать *не рекомендуется*, потому что в некоторых операционных системах пользователи могут просматривать аргументы вашей командной строки, что позволит им воспользоваться вашей учетной записью. Ключ `/NOLOG` в многопользовательских системах позволяет запустить SQL*Plus без подключения к базе данных. Имя пользователя и пароль задаются в команде `CONNECT`:

```
OS> sqlplus /nolog
```

```
SQL*Plus: Release 11.1.0.6.0 - Production on Fri Nov 7 10:28:26 2008
```

```
Copyright (c) 1982, 2007, Oracle. All rights reserved.
```

```
SQL> CONNECT bob/swordfish
```

```
SQL> Connected.
```

Если компьютер, на котором работает SQL*Plus, также содержит правильно сконфигурированное приложение Oracle Net¹ и вы авторизованы администратором для подключения к удаленным базам данных (то есть серверам баз данных, работающим на других компьютерах), то сможете подключаться к ним из SQL*Plus. Для этого наряду с именем пользователя и паролем необходимо ввести *идентификатор подключения* Oracle Net, называемый также *именем сервиса*. Идентификатор подключения может выглядеть так:

```
hqhr.WORLD
```

¹ Oracle Net — современное название продукта, который ранее назывался Net8 или SQL*Net.

Идентификатор вводится после имени пользователя и пароля, отделяясь от них символом «@»:

```
SQL> CONNECT bob/swordfish@hqhr.WORLD
SQL> Connected.
```

При запуске псевдографической версии SQL*Plus идентификационные данные вводятся в поле Host String (рис. 2.2). Если вы подключаетесь к серверу базы данных, работающему на локальной машине, оставьте поле пустым.

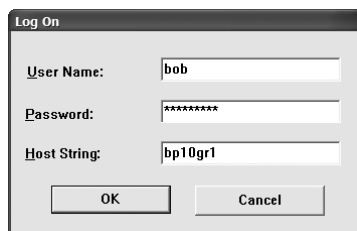


Рис. 2.2. Окно ввода идентификационных данных в SQL*Plus

После запуска SQL*Plus в программе можно делать следующее:

- выполнять SQL-инструкции;
- компилировать и сохранять программы на языке PL/SQL в базе данных;
- запускать программы на языке PL/SQL;
- выполнять команды SQL*Plus;
- запускать сценарии, содержащие сразу несколько перечисленных команд.

Рассмотрим поочередно каждую из перечисленных возможностей.

Выполнение SQL-инструкции

По умолчанию команды SQL в SQL*Plus завершаются символом «;» (точка с запятой), но вы можете сменить этот символ.

В консольной версии SQL*Plus запрос

```
SELECT isbn, author, title FROM books;
```

выдает результат, подобный тому, который показан на рис. 2.1¹.

Запуск программы на языке PL/SQL

Итак, приступаем. Введите в SQL*Plus небольшую программу на PL/SQL:

```
SQL> BEGIN
2   DBMS_OUTPUT.PUT_LINE('У меня получилось!');
3 END;
4 /
```

После ее выполнения экран выглядит так:

```
PL/SQL procedure successfully completed.
```

```
SQL>
```

¹ Здесь я немного смухлевал: для получения результатов в таком виде нужно воспользоваться командами форматирования столбцов. Если бы эта книга была посвящена SQL*Plus или возможностям вывода данных, то я бы описал разнообразные средства управления выводом. Но вам придется поверить мне на слово: этих средств больше, чем вы можете себе представить.

Странно — наша программа должна была вызвать встроенную программу PL/SQL, которая выводит на экран заданный текст. Однако SQL*Plus по умолчанию почему-то подавляет такой вывод. Чтобы увидеть выводимую программой строку, необходимо выполнить специальную команду SQL*Plus — `SERVEROUTPUT`:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE('У меня получилось!');
  3   END;
  4   /
```

И только теперь на экране появляется ожидаемая строка:

У меня получилось!

PL/SQL procedure successfully completed.

SQL>

Обычно я включаю команду `SERVEROUTPUT` в свой файл запуска (см. раздел «Автоматическая загрузка пользовательского окружения при запуске»). В таком случае она остается активной до тех пор, пока не произойдет одно из следующих событий:

- разрыв соединения, выход из системы или завершение сеанса с базой данных по иной причине;
- явное выполнение команды `SERVEROUTPUT` с атрибутом `OFF`;
- удаление состояния сеанса Oracle по вашему запросу или из-за ошибки компиляции;
- в версиях до Oracle9i Database Release 2 — ввод новой команды `CONNECT`. В последующих версиях SQL*Plus автоматически заново обрабатывает файл запуска после каждой команды `CONNECT`.

При вводе в консольном или псевдографическом приложении SQL*Plus команды SQL или PL/SQL программа назначает каждой строке, начиная со второй, порядковый номер. Нумерация строк используется по двум причинам: во-первых, она помогает вам определить, какую строку редактировать с помощью встроенного строкового редактора, а во-вторых, чтобы при обнаружении ошибки в вашем коде в сообщении об ошибке был указан номер строки. Вы еще не раз увидите *эту* возможность в действии.

Ввод команд PL/SQL в SQL*Plus завершается символом косой черты (строка 4 в приведенном примере). Этот символ обычно безопасен, но у него есть несколько важных характеристик:

- Косая черта означает, что введенную команду следует выполнить независимо от того, была это команда SQL или PL/SQL.
- Косая черта — это команда SQL*Plus; она *не является* элементом языка SQL или PL/SQL.
- Она должна находиться в отдельной строке, не содержащей никаких других команд.
- В большинстве версий SQL*Plus до Oracle9i косая черта, перед которой стоял один или несколько пробелов, не работала! Начиная с Oracle9i, среда SQL*Plus правильно интерпретирует начальные пробелы, то есть попросту игнорирует их. Завершающие пробелы игнорируются во всех версиях.

Для удобства SQL*Plus предлагает пользователям PL/SQL применять команду `EXECUTE`, которая позволяет не вводить команды `BEGIN`, `END` и завершающую косую черту. Таким образом, следующая строка эквивалентна приведенной выше программе:

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE('У меня получилось!')
```

Завершающая точка с запятой не обязательна, лично я предпочитаю ее опустить. Как и большинство других команд SQL*Plus, команду EXECUTE можно сократить, причем она не зависит от регистра символов. Поэтому указанную строку проще всего ввести так:

```
SQL> EXEC dbms_output.put_line('У меня получилось!')
```

Запуск сценария

Практически любую команду, которая может выполняться в интерактивном режиме SQL*Plus, можно сохранить в файле для повторного выполнения. Для выполнения сценария проще всего воспользоваться командой SQL*Plus @¹. Например, следующая конструкция выполняет все команды в файле abc.pkg:

```
SQL> @abc.pkg
```

Файл сценария должен находиться в текущем каталоге (или быть указанным в переменной SQLPATH).

Если вы предпочитаете имена команд, используйте эквивалентную команду START:

```
SQL> START abc.pkg
```

и вы получите идентичные результаты. В любом случае команда заставляет SQL*Plus выполнить следующие операции:

1. Открыть файл с именем abc.pkg.
2. Последовательно выполнить все команды SQL, PL/SQL и SQL*Plus, содержащиеся в указанном файле.
3. Закрыть файл и вернуть управление в приглашение SQL*Plus (если в файле нет команды EXIT, выполнение которой завершает работу SQL*Plus).

Например:

```
SQL> @abc.pkg
```

```
Package created.
```

```
Package body created.
```

```
SQL>
```

По умолчанию SQL*Plus выводит на экран только результаты выполнения команд. Если вы хотите увидеть исходный код файла сценария, используйте команду SQL*Plus SET ECHO ON.

В приведенном примере используется файл с расширением pkg. Если указать имя файла без расширения, произойдет следующее:

```
SQL> @abc
SP2-0310: unable to open file "abc.sql"
```

Как видите, по умолчанию предполагается расширение sql. Здесь «SP2-0310» — код ошибки Oracle, а «SP2» означает, что ошибка относится к SQL*Plus. (За дополнительной информацией о сообщениях ошибок SQL*Plus обращайтесь к руководству Oracle «SQL*Plus User's Guide and Reference».)

¹ Команды START, @ и @@ доступны в небраузерной версии SQL*Plus. В iSQL*Plus для получения аналогичных результатов используются кнопки Browse и Load Script.

Что такое «текущий каталог»?

При запуске SQL*Plus из командной строки операционной системы SQL*Plus использует текущий каталог операционной системы в качестве своего текущего каталога. Иначе говоря, если запустить SQL*Plus командой

```
C:\BOB\FILES> sqlplus
```

все операции с файлами в SQL*Plus (такие, как открытие файла или запуск сценария) по умолчанию будут выполняться с файлами каталога C:\BOB\FILES.

Если SQL*Plus запускается ярлыком или командой меню, то текущим каталогом будет каталог, который ассоциируется операционной системой с механизмом запуска. Как же сменить текущий каталог из SQL*Plus? Ответ зависит от версии. В консольной программе это просто невозможно: вы должны выйти из программы, изменить каталог средствами операционной системы и перезапустить SQL*Plus. В версии с графическим интерфейсом у команды меню File ► Open или File ► Save имеется побочный эффект: она меняет текущий каталог.

Если файл сценария находится в другом каталоге, то перед именем файла следует указать путь к нему¹:

```
SQL> @/files/src/release/1.0/abc.pkg
```

С запуском сценария из другого каталога связан интересный вопрос: что, если файл abc.pkg расположен в другом каталоге и, в свою очередь, запускает другие сценарии? Например, он может содержать такие строки:

```
REM имя файла: abc.pkg
@@abc.pks
@@abc.pkb
```

(Любая строка, начинающаяся с ключевого слова REM, является комментарием, и SQL*Plus ее игнорирует.) Предполагается, что сценарий abc.pkg будет вызывать сценарии abc.pks и abc.pkb. Но если информация о пути отсутствует, где же SQL*Plus будет их искать?

```
C:\BOB\FILES> sqlplus
```

```
...
SQL> @/files/src/release/1.0/abc.pkg
SP2-0310: unable to open file "abc.pks"
SP2-0310: unable to open file "abc.pkb"
```

Оказывается, поиск выполняется только в каталоге, из которого был запущен исходный сценарий. Для решения данной проблемы существует команда @@. Она означает, что в качестве текущего каталога должен временно рассматриваться каталог, в котором находится выполняемый файл. Таким образом, команды запуска в сценарии abc.pkg следует записывать так:

```
REM имя файла: abc.pkg
@@@abc.pks
@@@abc.pkb
```

Теперь результат выглядит иначе:

```
C:\BOB\FILES> sqlplus
...
SQL> @/files/src/release/1.0/abc.pkg
```

```
Package created.
```

```
Package body created.
```

```
...как, собственно, и было задумано.
```

¹ Косая черта может использоваться в качестве разделителей каталогов как в Unix/Linux, так и в операционных системах Microsoft. Это упрощает перенос сценариев между операционными системами.

Другие задачи SQL*Plus

SQL*Plus поддерживает десятки команд, но мы остановимся лишь на некоторых из них, самых важных или особенно малопонятных для пользователя. Для более обстоятельного изучения продукта следует обратиться к книге Джонатана Генника Oracle SQL*Plus: The Definitive Guide (издательство O'Reilly), а за краткой справочной информацией — к книге Oracle SQL*Plus Pocket Reference того же автора.

Пользовательские установки

Многие аспекты поведения SQL*Plus могут быть изменены при помощи встроенных переменных и параметров. Один из примеров такого рода уже встречался нам при применении выражения SET SERVEROUTPUT. Команда SQL*Plus SET также позволяет задавать многие другие настройки. Так, выражение SET SUFFIX изменяет используемое по умолчанию расширение файла, а SET LINESIZE *n* — задает максимальное количество символов в строке (символы, не помещающиеся в строке, переносятся в следующую). Сводка всех SET-установок текущего сеанса выводится командой

```
SQL> SHOW ALL
```

Приложение SQL*Plus также позволяет создавать собственные переменные в памяти и задавать специальные переменные, посредством которых можно управлять его настройками. Переменные SQL*Plus делятся на два вида: DEFINE и переменные привязки. Значение DEFINE-переменной задается командой DEFINE:

```
SQL> DEFINE x = "ответ 42"
```

Чтобы просмотреть значение *x*, введите следующую команду:

```
SQL> DEFINE x
DEFINE X = "ответ 42" (CHAR)
```

Ссылка на данную переменную обозначается символом &. Перед передачей инструкции Oracle SQL*Plus выполняет простую подстановку, поэтому если значение переменной должно использоваться как строковый литерал, ссылку следует заключить в кавычки:

```
SELECT '&x' FROM DUAL;
```

Переменная привязки объявляется командой VARIABLE. В дальнейшем ее можно будет использовать в PL/SQL и вывести значение на экран командой SQL*Plus PRINT:

```
SQL> VARIABLE x VARCHAR2(10)
SQL> BEGIN
2      :x := 'hullo';
3  END;
4  /
```

PL/SQL procedure successfully completed.

```
SQL> PRINT :x
```

```
x
-----
hullo
```

Ситуация немного запутывается, потому что у нас теперь две разные переменные *x*: одна определяется командой DEFINE, а другая — командой VARIABLE:

```
SQL> SELECT :x, '&x' FROM DUAL;
old  1: SELECT :x, '&x' FROM DUAL
new  1: SELECT :x, 'ответ 42' FROM DUAL

:x                                'OTBET42'
-----
hullo                             ответ 42
```

Запомните, что **DEFINE**-переменные всегда представляют собой символьные строки, которые SQL*Plus заменяет их текущими значениями, а **VARIABLE**-переменные используются в SQL и PL/SQL как настоящие переменные привязки.

Сохранение результатов в файле

Выходные данные сеанса SQL*Plus часто требуется сохранить в файле — например, если вы строите отчет, или хотите сохранить сводку своих действий на будущее, или динамически генерируете команды для последующего выполнения. Все эти задачи легко решаются в SQL*Plus командой **SPPOOL**:

```
SQL> SPOOL report
SQL> @run_report
...выходные данные выводятся на экран и записываются в файл report.lst...
SQL> SPOOL OFF
```

Первая команда **SPPOOL** сообщает SQL*Plus, что все строки данных, выводимые после нее, должны сохраняться в файле **report.lst**. Расширение **lst** используется по умолчанию, но вы можете переопределить его, указывая нужное расширение в команде **SPPOOL**:

```
SQL> SPOOL report.txt
```

Вторая команда **SPPOOL** приказывает SQL*Plus прекратить сохранение результатов и закрыть файл.

Выход из SQL*Plus

Чтобы выйти из SQL*Plus и вернуться в операционную систему, выполните команду **EXIT**:

```
SQL> EXIT
```

Если в момент выхода из приложения данные записывались в файл, SQL*Plus прекращает запись и закрывает файл.

А что произойдет, если в ходе сеанса вы внесли изменения в данные некоторых таблиц, а затем вышли из SQL*Plus без явного завершения транзакции? По умолчанию SQL*Plus принудительно закрепляет незавершенные транзакции, если только сеанс не завершился с ошибкой SQL или если вы не выполнили команду SQL*Plus **WHenever SQLERROR EXIT ROLLBACK** (см. далее раздел «Обработка ошибок в SQL*Plus»).

Чтобы разорвать подключение к базе данных, но остаться в SQL*Plus, следует выполнить команду **CONNECT**. Результат ее выполнения выглядит примерно так:

```
SQL> DISCONNECT
Disconnected from Personal Oracle Database 10g Release 10.1.0.3.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

Для смены подключений команда **DISCONNECT** не обязательна — достаточно ввести команду **CONNECT**, и SQL*Plus автоматически разорвет первое подключение перед созданием второго. Тем не менее команда **DISCONNECT** вовсе не лишняя — при использовании средств аутентификации операционной системы¹ сценарий может автоматически восстановить подключение... к чужой учетной записи. Я видел подобные примеры.

Редактирование инструкции

SQL*Plus хранит последнюю выполненную инструкцию в буфере. Содержимое буфера можно отредактировать во встроенном редакторе либо в любом внешнем редакторе по вашему выбору. Начнем с процесса настройки и использования внешнего редактора.

¹ Аутентификация операционной системы позволяет запускать SQL*Plus без ввода имени пользователя и пароля.

Команда `EDIT` сохраняет буфер в файле, временно приостанавливает выполнение SQL*Plus и передает управление редактору:

SQL> `EDIT`

По умолчанию файл будет сохранен под именем `afiedt.buf`, но вместо этого имени можно выбрать другое (команда `SET EDITFILE`). Если же вы хотите отредактировать существующий файл, укажите его имя в качестве аргумента `EDIT`:

SQL> `EDIT abc.pkg`

После сохранения файла и выхода из редактора сеанс SQL*Plus читает содержимое отредактированного файла в буфер, а затем продолжает работу.

По умолчанию Oracle использует следующие внешние редакторы:

- `ed` в Unix, Linux и других системах этого семейства;
- Блокнот в системах Microsoft Windows.

Хотя выбор редактора по умолчанию жестко запрограммирован в исполняемом файле `sqlplus`, его легко изменить, присвоив переменной `_EDITOR` другое значение. Например, я часто использую следующую команду:

SQL> `DEFINE _EDITOR = /bin/vi`

Здесь `/bin/vi` — полный путь к редактору, хорошо известному в среде «технарей». Я рекомендую задавать полный путь к редактору по соображениям безопасности.

Если же вы хотите работать со встроенным строковым редактором SQL*Plus (иногда это в самом деле *бывает* удобно), вам необходимо знать следующие команды:

- `L` — вывод последней команды.
- `n` — сделать текущей строкой `n`-ю строку команды.
- `DEL` — удалить текущую строку.
- `C/old/new/` — заменить первое вхождение `old` в текущей строке на `new` (разделителем может быть произвольный символ, в данном случае это косая черта).
- `n text` — сделать `text` текущим текстом строки `n`.
- `I` — вставить строку после текущей. Чтобы вставить новую строку перед первой, используйте команду с нулевым номером строки (например, `0 text`).

Автоматическая загрузка пользовательского окружения при запуске

Для настройки среды разработки SQL*Plus можно изменять один или оба сценария ее запуска. SQL*Plus при запуске выполняет две основные операции:

- 1) ищет в корневом каталоге Oracle файл `sqlplus/admin/glogin.sql` и выполняет содержащиеся в нем команды (этот «глобальный» сценарий реализуется независимо от того, кто запустил SQL*Plus из корневого каталога Oracle и какой каталог был при этом текущим);
- 2) находит и выполняет файл `login.sql` в текущем каталоге, если он существует¹.

Начальный сценарий может содержать те же типы команд, что и любой другой сценарий SQL*Plus: команды `SET`, SQL-инструкции, команды форматирования столбцов и т. д.

Ни один из файлов не является обязательным. Если присутствуют оба файла, то сначала выполняется `glogin.sql`, а затем `login.sql`; в случае конфликта настроек или переменных преимущество получают установки последнего файла, `login.sql`.

¹ А если не существует, но переменная `SQLPATH` содержит один или несколько каталогов, разделенных двоеточиями, SQL*Plus просматривает эти каталоги и выполняет первый обнаруженный файл `login.sql`.

Несколько полезных установок в файле `login.sql`:

```
REM Количество строк выходных данных инструкции SELECT
REM перед повторным выводом заголовков
SET PAGESIZE 999

REM Ширина выводимой строки в символах
SET LINESIZE 132

REM Включение вывода сообщений DBMS_OUTPUT.
REM В версиях, предшествующих Oracle Database 10g Release 2,
REM вместо UNLIMITED следует использовать значение 1000000.
SET SERVEROUTPUT ON SIZE UNLIMITED FORMAT WRAPPED

REM Замена внешнего текстового редактора
DEFINE _EDITOR = /usr/local/bin/vim

REM Форматирование столбцов, извлекаемых из словаря данных
COLUMN segment_name FORMAT A30 WORD_WRAP
COLUMN object_name FORMAT A30 WORD_WRAP

REM Настройка приглашения (работает в SQL*Plus Oracle9i и выше)
SET SQLPROMPT "_USER'@'_CONNECT_IDENTIFIER > "
```

Обработка ошибок в SQL*Plus

Способ, которым SQL*Plus информирует вас об успешном завершении операции, зависит от класса выполняемой команды. Для большинства команд SQL*Plus признаком успеха является отсутствие сообщений об ошибках. С другой стороны, успешное выполнение инструкций SQL и PL/SQL обычно сопровождается выдачей какой-либо текстовой информации.

Если ошибка содержится в команде SQL или PL/SQL, SQL*Plus по умолчанию сообщает о ней и продолжает работу. Это удобно в интерактивном режиме, но при выполнении сценария желательно, чтобы при возникновении ошибки работа SQL*Plus прерывалась. Для этого применяется команда

```
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE
```

SQL*Plus прекратит работу, если после выполнения команды сервер базы данных в ответ на команду SQL или PL/SQL вернет сообщение об ошибке. Часть приведенной выше команды, `SQL.SQLCODE`, означает, что при завершении работы SQL*Plus установит ненулевой код завершения, значение которого можно проверить на стороне вызова¹. В противном случае SQL*Plus всегда завершается с кодом 0, что может быть неверно истолковано как успешное выполнение сценария.

Другая форма указанной команды:

```
SQL> WHENEVER SQLERROR SQL.SQLCODE EXIT ROLLBACK
```

означает, что перед завершением SQL*Plus будет произведен откат всех несохраненных изменений данных.

Достоинства и недостатки SQL*Plus

Кроме тех, что указаны выше, у SQL*Plus имеется несколько дополнительных функций, которые вам наверняка пригодятся.

- С помощью SQL*Plus можно запускать пакетные программы, задавая в командной строке аргументы и обращаясь к ним по ссылкам вида `&1` (первый аргумент), `&2` (второй аргумент) и т. д.

¹ Например, с помощью системной переменной `$?` в Unix и `%ERRORLEVEL%` в Microsoft Windows.

- SQL*Plus обеспечивает полную поддержку всех команд SQL и PL/SQL. Это обычно имеет значение при использовании специфических возможностей Oracle. В средах сторонних разработчиков отдельные элементы указанных языков могут поддерживаться не в полном объеме. Например, некоторые из них до сих пор не поддерживают объектные типы Oracle, введенные несколько лет назад.
- SQL*Plus работает на том же оборудовании и в тех же операционных системах, что и сервер Oracle.

Как и любые другие инструментальные средства, SQL*Plus имеет свои недостатки:

- В консольных версиях SQL*Plus буфер команд содержит только последнюю из ранее использовавшихся команд. Журнала команд эта программа не ведет.
- SQL*Plus не имеет таких возможностей, характерных для современных интерпретаторов команд, как автоматическое завершение ключевых слов и подсказки о доступных объектах базы данных, появляющиеся при вводе команд.
- Электронная справка содержит минимальную информацию о наборе команд SQL*Plus. (Для получения справки по конкретной команде используется команда HELP.)
- После запуска SQL*Plus сменить текущий каталог невозможно, что довольно неудобно, если вы часто открываете и сохраняете сценарии и при этом не хотите постоянно указывать полный путь к файлу. Если вы обнаружили, что находитесь не в том каталоге, вам придется выйти из SQL*Plus, сменить каталог и снова запустить программу.
- Если не использовать механизм SQLPATH, который я считаю потенциально опасным, SQL*Plus ищет файл login.sql только в каталоге запуска; было бы лучше, если бы программа продолжала поиск файла в домашнем каталоге стартового сценария.

Итак, SQL*Plus не отличается удобством в работе и изысканностью интерфейса. Но данное приложение используется повсеместно, работает надежно и наверняка будет поддерживаться до тех пор, пока существует Oracle Corporation.

Базовые операции PL/SQL

Теперь поговорим о создании, выполнении, удалении и других операциях с программами PL/SQL, выполняемыми с помощью SQL*Plus. В этом разделе мы обойдемся без лишних подробностей; скорее, он представляет собой вводный обзор тем, которые будут подробно рассматриваться в следующих главах.

Создание хранимой программы

Для того чтобы написать собственную программу на PL/SQL, нужно воспользоваться одной из инструкций SQL CREATE. Например, если вы хотите создать хранимую функцию именем wordcount для подсчета количества слов в строке, выполните инструкцию CREATE FUNCTION:

```
CREATE FUNCTION wordcount (str IN VARCHAR2)
  RETURN PLS_INTEGER
AS
  здесь объявляются локальные переменные
BEGIN
  здесь реализуется алгоритм
END;
/
```

Как и в случае с простыми блоками BEGIN-END, приводившимися ранее, код этой инструкции в SQL*Plus должен завершаться символом косой черты, который размещается в отдельной строке.

Если администратор базы данных предоставил вам привилегию создания процедур `CREATE PROCEDURE` (которая также включает привилегию создания функций), эта инструкция заставит Oracle откомпилировать и сохранить в схеме заданную хранимую функцию. И если код будет откомпилирован успешно, вы увидите следующее сообщение:

Function created.

Если в схеме Oracle уже имеется объект (таблица или пакет) с именем `wordcount`, выполнение инструкции `CREATE FUNCTION` завершится сообщением об ошибке:

ORA-00955: name is already used by an existing object.

По этой причине Oracle поддерживает инструкцию `CREATE OR REPLACE FUNCTION` — вероятно, вы будете использовать ее в 99 случаях из 100:

```
CREATE OR REPLACE FUNCTION wordcount (str IN VARCHAR2)
RETURN PLS_INTEGER
AS то же, что в приведенном выше примере
```

Связка `OR REPLACE` позволяет избежать побочных эффектов, вызванных удалением и повторным созданием программ; она сохраняет все привилегии на объект, предоставленные другим пользователям или ролям. При этом она заменяет только объекты одного типа и не станет автоматически удалять таблицу с именем `wordcount` только потому, что вы решили создать функцию с таким же именем.

Программисты обычно сохраняют подобные команды (равно как и анонимные блоки, предназначенные для повторного использования) в файлах операционной системы. Например, для хранения рассматриваемой функции можно было бы создать файл `wordcount.fun`, а для его запуска применить команду SQL*Plus @:

```
SQL> @wordcount.fun
```

Function created.

Как упоминалось ранее, SQL*Plus по умолчанию не выводит содержимое сценария на экран. Для того чтобы исходный код сценария, включая присвоенные Oracle номера строк, отображался на экране, воспользуемся командой `SET ECHO ON`. Особенно полезна эта команда в ходе диагностики. Давайте намеренно допустим в программе ошибку, закомментировав объявление переменной:

```
SQL> /* Файл в Сети: wordcount.fun */
SQL> SET ECHO ON
SQL> @wordcount.fun
SQL> CREATE OR REPLACE FUNCTION wordcount (str IN VARCHAR2)
2     RETURN PLS_INTEGER
3 AS
4 /* words PLS_INTEGER := 0; ***Намеренное внесение ошибки*** */
5     len PLS_INTEGER := NVL(LENGTH(str),0);
6     inside_a_word BOOLEAN;
7 BEGIN
8     FOR i IN 1..len + 1
9     LOOP
10        IF ASCII(SUBSTR(str, i, 1)) < 33 OR i > len
11        THEN
12            IF inside_a_word
13            THEN
14                words := words + 1;
15                inside_a_word := FALSE;
16            END IF;
17        ELSE
18            inside_a_word := TRUE;
19        END IF;
20    END LOOP;
```

продолжение ➤

```

21     RETURN words;
22 END;
23 /

```

Warning: Function created with compilation errors.

SQL>

Предупреждение сообщает нам о том, что функция была создана, но из-за ошибок компиляции ее выполнение невозможно. Нам удалось сохранить исходный код в базе данных; теперь нужно извлечь подробную информацию об ошибке из базы данных. Проще всего это сделать с помощью команды SQL*Plus SHOW ERRORS, которую можно сократить до SHO ERR:

SQL> SHO ERR

Errors for FUNCTION WORDCOUNT:

LINE/COL ERROR

```

-----
14/13      PLS-00201: identifier 'WORDS' must be declared
14/13      PL/SQL: Statement ignored
21/4       PL/SQL: Statement ignored
21/11      PLS-00201: identifier 'WORDS' must be declared

```

ВЫВОД ДРУГИХ ОШИБОК

Многие программисты Oracle знают только одну форму команды SQL*Plus:

SQL> SHOW ERRORS

Они ошибочно полагают, что для получения дополнительной информации об ошибках, не встречавшихся при последней компиляции, необходимо обращаться с запросом к представлению USER_ERRORS. Однако если указать в команде SHOW ERRORS категорию и имя объекта, вы получите информацию о последних связанных с ним ошибках:

SQL> SHOW ERRORS категория [схема.]объект

Например, чтобы просмотреть информацию о последних ошибках в процедуре wordcount, выполните такую команду:

SQL> SHOW ERRORS FUNCTION wordcount

Будьте внимательны при интерпретации выходного сообщения:

No errors.

Оно выводится в трех случаях: (1) когда код объекта откомпилирован успешно; (2) вы задали неверную категорию (скажем, функцию вместо процедуры); и (3) объект с заданным именем не существует.

Полный список категорий, поддерживаемых этой командой, зависит от версии СУБД, но в него как минимум входят следующие категории:

```

DIMENSION
FUNCTION
JAVA SOURCE
JAVA CLASS
PACKAGE
PACKAGE BODY
PROCEDURE
TRIGGER
TYPE
TYPE BODY
VIEW

```

Компилятор обнаружил оба вхождения переменной и сообщил точные номера строк и столбцов. Более подробную информацию об ошибке можно найти по идентификатору (в данном случае PLS-00201) в документации Oracle *Database Error Messages*.

Во внутренней реализации команда `SHOW ERRORS` обращается с запросом к представлению Oracle `USER_ERRORS` из словаря данных. В принципе вы можете обращаться к этому представлению и самостоятельно, но обычно это просто не нужно (см. врезку «Вывод других ошибок»).

Команда `SHOW ERRORS` часто добавляется послед каждой инструкции `CREATE`, создающей хранимую программу PL/SQL. Поэтому типичный шаблон для построения хранимых программ в SQL*Plus может начинаться так:

```
CREATE OR REPLACE тип_программы
AS
    ваш код
END;
/
```

`SHOW ERRORS`

(Обычно я не включаю команду `SET ECHO ON` в сценарий, а просто ввожу ее в командной строке, когда это потребуется.)

Если ваша программа содержит ошибку, которая может быть обнаружена компилятором, инструкция `CREATE` сохранит эту программу в базе данных, но в нерабочем состоянии. Если же вы неверно используете синтаксис `CREATE`, то Oracle не поймет, что вы пытаетесь сделать, и не сохранит код в базе данных.

Выполнение хранимой программы

Мы рассмотрели два способа вызова хранимой программы: заключение ее в простом блоке PL/SQL и использование команды `EXECUTE` среды SQL*Plus. Одни хранимые программы также можно использовать в других. Например, функция `wordcount` может использоваться в любом месте, где может использоваться целочисленное выражение. Короткий пример тестирования функции `wordcount` с входным значением `CHR(9)`, которое является ASCII-кодом символа табуляции:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Введенная строка содержит ' || wordcount(CHR(9)) || '
слов');
END;
/
```

Вызов функции `wordcount` включен в выражение как аргумент процедуры `DBMS_OUTPUT.PUT_LINE`. В таких случаях PL/SQL автоматически преобразует целое число в строку, чтобы соединить его с двумя другими литеральными выражениями. Результат получается следующим:

Введенная строка содержит 0 слов

Многие функции PL/SQL можно вызывать и из SQL-инструкций. Несколько примеров использования функции `wordcount`:

- Включение в список выборки для вычисления количества слов в столбце таблицы:
`SELECT isbn, wordcount(description) FROM books;`
- Использование в ANSI-совместимой инструкции `CALL` для привязки выходных данных функции к переменной SQL*Plus и вывода результата:
`VARIABLE words NUMBER`
`CALL wordcount('некоторый_ текст') INTO :words;`
`PRINT :words`

То же, но с выполнением функции из удаленной базы данных, определяемой ссылкой `test.newyork.ora.com`:

```
CALL wordcount@test.newyork.ora.com('некоторый_текст') INTO :words;
```

Выполнение функции, принадлежащей схеме `bob`, при подключении к любой схеме с соответствующей привилегией:

```
SELECT bob.wordcount(description) FROM books WHERE id = 10007;
```

Вывод хранимых программ

Рано или поздно вам потребуется просмотреть список имеющихся хранимых программ и последние версии их исходного кода, которые Oracle хранит в словаре данных. Эту задачу намного проще выполнить в графических служебных программах, но если у вас такой программы нет, можно написать несколько SQL-инструкций, извлекающих из словаря данных нужную информацию.

Так, чтобы просмотреть полный список программ (а также таблиц, индексов и других элементов), запросите информацию представления `USER_OBJECTS`:

```
SELECT * FROM USER_OBJECTS;
```

Представление содержит сведения о каждом объекте: его имя, тип, время создания, время последней компиляции, состояние работоспособности и другую полезную информацию.

Если вам нужно получить данные об интерфейсе программы в SQL*Plus, проще всего воспользоваться командой `DESCRIBE`:

```
SQL> DESCRIBE wordcount
```

| Function Name | Returns | Binary_Integer | Type | In/Out | Default? |
|---------------|---------|----------------|----------|--------|----------|
| STR | | | VARCHAR2 | IN | |

Команда `DESCRIBE` также работает с таблицами, объектными типами, процедурами и пакетами. Чтобы просмотреть полный исходный код хранимых программ, обратитесь с запросом к представлению `USER_SOURCE` или `TRIGGER_SOURCE`. (О том, как получить информацию из этих представлений словаря данных, подробно рассказано в главе 20.)

Управление привилегиями и создание синонимов хранимых программ

Созданную вами программу на PL/SQL обычно не может выполнять никто, кроме вас или администратора базы данных. Предоставить право на ее применение другому пользователю можно с помощью инструкции `GRANT`:

```
GRANT EXECUTE ON wordcount TO scott;
```

Инструкция `REVOKE` лишает пользователя этой привилегии:

```
REVOKE EXECUTE ON wordcount FROM scott;
```

Привилегия выполнения `EXECUTE` также может быть представлена роли:

```
GRANT EXECUTE ON wordcount TO all_mis;
```

а также всем пользователям Oracle:

```
GRANT EXECUTE ON wordcount TO PUBLIC;
```

Если привилегия `EXECUTE` представляется отдельному пользователю (например, с идентификатором `scott`), затем — роли, в которую входит этот пользователь (например, `all_mis`), и наконец, — всем пользователям, Oracle запомнит все три варианта ее

предоставления. Любой из них позволит пользователю `scott` выполнять программу. Но если вы захотите лишить данного пользователя этой возможности, то сначала следует отменить привилегию пользователя с идентификатором `scott`, а затем аннулировать привилегию на выполнение функции для всех пользователей (`PUBLIC`) и роли (или же исключить пользователя из этой роли).

Для просмотра списка привилегий, предоставленных другим пользователям и ролям, можно запросить информацию представления `USER_TAB_PRIVS_MADE`. Имена программ в этом представлении почему-то выводятся в столбце `table_name`:

```
SQL> SELECT table_name, grantee, privilege
       2 FROM USER_TAB_PRIVS_MADE
       3 WHERE table_name = 'WORDCOUNT';
```

| TABLE_NAME | GRANTEE | PRIVILEGE |
|------------|---------|-----------|
| WORDCOUNT | PUBLIC | EXECUTE |
| WORDCOUNT | SCOTT | EXECUTE |
| WORDCOUNT | MIS_ALL | EXECUTE |

Если пользователь `scott` имеет привилегию `EXECUTE` на выполнение программы `wordcount`, он, возможно, захочет создать для нее синоним, чтобы ему не приходилось указывать перед именем программы префикс с именем схемы:

```
SQL> CONNECT scott/tiger
Connected.
SQL> CREATE OR REPLACE SYNONYM wordcount FOR bob.wordcount;
```

Теперь пользователь может выполнять программу, ссылаясь на ее синоним:

```
IF wordcount(localvariable) > 100 THEN...
```

Так удобнее, потому что в случае изменения владельца программы достаточно будет изменить только ее синоним, а не все те хранимые программы, из которых она вызывается.

Синоним можно определить для процедуры, функции, пакета или пользовательского типа. В синонимах процедур, функций и пакетов может скрываться не только схема, но и база данных; синонимы для удаленных программ создаются так же просто, как и для локальных. Однако синонимы могут скрывать только идентификаторы схем и баз данных; синоним не может использоваться вместо пакетной подпрограммы.

Созданный синоним удаляется простой командой:

```
DROP SYNONYM wordcount;
```

Удаление хранимой программы

Если вы твердо уверены в том, что какая-либо хранимая программа вам уже не понадобится, удалите ее с помощью команды `SQL DROP`. Например, следующая команда удаляет хранимую функцию `wordcount`:

```
DROP FUNCTION wordcount;
```

Полное удаление пакета, который может состоять из двух элементов (спецификации и тела):

```
DROP PACKAGE pkgname;
```

Также можно удалить только тело пакета без отмены соответствующей спецификации:

```
DROP PACKAGE BODY pkgname;
```

При удалении программы, которая вызывается из других программ, последние помечаются как недействительные (`INVALID`).

Соккрытие исходного кода хранимой программы

При создании программы PL/SQL описанным выше способом ее исходный код сохраняется в словаре данных в виде обычного текста, который администратор базы данных может просмотреть и даже изменить. Для сохранения профессиональных секретов и предотвращения постороннего вмешательства в программный код перед распространением его следует зашифровать или скрыть иным способом.

Oracle предлагает приложение командной строки `wrap`, которое преобразует серию команд `CREATE` в комбинацию обычного текста и шестнадцатеричных кодов. Это действие не является шифрованием в прямом смысле слова, но все же направлено на соккрытие кода. Приведем несколько фрагментов преобразованного кода:

```
FUNCTION wordcount wrapped
0
abcd
abcd
...разрыв...
1WORDS:
10:
1LEN:
1NVL:
1LENGTH:
1INSIDE_A_WORD:
1BOOLEAN:
...разрыв...
a5 b 81 b0 a3 a0 1c 81
b0 91 51 a0 7e 51 a0 b4
2e 63 37 :4 a0 51 a5 b a5
b 7e 51 b4 2e :2 a0 7e b4
2e 52 10 :3 a0 7e 51 b4 2e
d :2 a0 d b7 19 3c b7 :2 a0
d b7 :2 19 3c b7 a0 47 :2 a0
```

Но если вам понадобится полноценное шифрование (скажем, для передачи такой секретной информации, как пароль), полагаться на возможности `wrap` не следует¹.

За дополнительной информацией о программе `wrap` обращайтесь к главе 20.

Средства разработки для PL/SQL

Как упоминалось ранее, для редактирования и выполнения кода PL/SQL можно использовать простейшую среду типа SQL*Plus, но вы также можете выбрать интегрированную среду разработки с полнофункциональным графическим интерфейсом, который сделает вашу работу более эффективной. В этом разделе перечислены некоторые популярные интегрированные среды (IDE). Я не стану рекомендовать ту или иную среду; тщательно проанализируйте списки требований и приоритетов и выберите тот вариант, который лучше всего соответствует вашим потребностям.

| Продукт | Описание |
|---------------|---|
| Toad | Разработчик — Quest Software. Безусловно, Toad является самой популярной интегрированной средой для работы с PL/SQL. Эта среда используется сотнями тысяч разработчиков — как в бесплатной, так и в коммерческой версии |
| SQL Navigator | SQL Navigator (еще одна разработка Quest Software) тоже используется десятками тысяч разработчиков, по достоинству оценивших интерфейс и вспомогательные функции программы |

¹ Oracle предоставляет полноценные средства шифрования в приложениях — используйте встроенный пакет `DBMS_CRYPTO` (или `DBMS_OBFUSCATION_TOOLKIT` в версиях до Oracle10g). За информацией о `DBMS_CRYPTO` обращайтесь к главе 23.

| Продукт | Описание |
|------------------|---|
| PL/SQL Developer | Многие разработчики отдали предпочтение продукту, предлагаемому фирмой Allround Automations. Он построен на базе модульной архитектуры, что позволяет сторонним разработчикам создавать собственные расширения для базового продукта |
| SQL Developer | Компания Oracle Corporation, много лет не предлагавшая практически никаких средств для редактирования PL/SQL, выпустила SQL Developer как «побочную ветвь» инструментария JDeveloper. SQL Developer распространяется бесплатно, а возможности этой программы непрерывно расширяются |

Также существует немало других интегрированных сред для работы с PL/SQL, но самые лучшие и популярные среды перечислены в таблице.

Вызов кода PL/SQL из других языков

Рано или поздно вам захочется вызвать код PL/SQL из кода C, Java, Perl, PHP или другого языка. Потребность вполне естественная, но если вам когда-нибудь приходилось создавать приложения с межъязыковыми вызовами, то вы наверняка сталкивались с различными нюансами преобразования типов данных разных языков (в первую очередь составных типов — массивов, записей, объектов и т. д.), не говоря уже о различиях в семантике параметров или специфичных для конкретного производителя расширений «стандартных» API, в частности Microsoft ODBC.

Рассмотрим несколько примеров использования PL/SQL во внешнем коде. Допустим, у нас имеется функция PL/SQL, которая получает код ISBN в строковом формате и возвращает заголовок соответствующей ему книги:

```
/* Файл в Сети: booktitle.fun */
FUNCTION booktitle (isbn_in IN VARCHAR2)
RETURN VARCHAR2
IS
    l_title books.title%TYPE;
    CURSOR icur IS SELECT title FROM books WHERE isbn = isbn_in;
BEGIN
    OPEN icur;
    FETCH icur INTO l_title;
    CLOSE icur;
    RETURN l_title;
END;
```

В SQL*Plus существует несколько способов вызова этой функции. Простейший способ:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(booktitle('0-596-00180-0'))
Learning Oracle PL/SQL
```

PL/SQL procedure successfully completed.

Давайте посмотрим, как вызвать эту функцию в следующих средах:

- C, с использованием прекомпилятора Oracle (Pro*C);
- Java, с использованием JDBC;
- Perl, с использованием интерфейсного модуля Perl DBI и драйвера DBD::Oracle;
- PHP;
- PL/SQL Server Pages.

Учтите, что примеры написаны исключительно в учебных целях — например, имя пользователя и пароль в них жестко закодированы, и программы просто направляют результат в стандартный выходной поток. Более того, я даже не стану подробно описывать каждую строку кода. Тем не менее эти примеры дают некоторое представление о том, какие схемы вызова PL/SQL могут использоваться в разных языках.

С, с использованием прекомпилятора Oracle (Pro*C)

Oracle предоставляет как минимум два разных внешних интерфейса для языка С: один называется OCI (Oracle Call Interface) и ориентируется в основном на программистов экстра-класса, а другой — Pro*C. В состав OCI входит множество функций для выполнения таких низкоуровневых операций, как открытие и разбор данных, привязка, выполнение, выборка... и все это для реализации лишь одного запроса! Поскольку простейшая программа на базе OCI, выполняющая не самую сложную задачу, содержит около 200 строк кода, мы рассмотрим пример использования Pro*C. Pro*C базируется на технологии прекомпиляции, позволяющей совмещать в исходном коде операторы языков С, SQL и PL/SQL. Такие файлы обрабатываются программой Oracle `procs`, которая преобразует их в код на языке С:

```
/* Файл в Сети: callbooktitle.pc */
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR uid[20];
    VARCHAR pwd[20];
    VARCHAR isbn[15];
    VARCHAR btitle[400];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA.H;

int sqlerror();

int main()
{
    /* Значения VARCHAR преобразуются в структуры из символьного массива и текущей
    длины */
    strcpy((char *)uid.arr,"scott");
    uid.len = (short) strlen((char *)uid.arr);
    strcpy((char *)pwd.arr,"tiger");
    pwd.len = (short) strlen((char *)pwd.arr);

    /* Гибрид исключения и оператора goto */
    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    /* Подключение к серверу Oracle с выполнением функции booktitle */
    EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;

    EXEC SQL EXECUTE
        BEGIN
            :btitle := booktitle('0-596-00180-0');
        END;
    END-EXEC;

    /* Вывод результата */
    printf("%s\n", btitle.arr);

    /* Отключение от сервера ORACLE. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

Как видите, Pro*С вряд ли приведет в восторг блюстителей чистоты языка С. Однако во многих компаниях считают, что Pro*С (или Pro*Cobol, или прекомпиляторы для других поддерживаемых Oracle языков программирования) может служить разумной альтернативой Visual Basic (слишком медленный и неуклюжий) и OCI (слишком сложный).

Лучшим источником информации о Pro*С является документация Oracle.

Java, с использованием JDBC

Как и в случае с С, Oracle поддерживает несколько разных методов подключения к базе данных из кода Java. Один из них — встроенный синтаксис SQL, называемый SQLJ — близок к другим прекомпиляторным технологиям Oracle, хотя и более удобен для отладки. Большей популярностью пользуется метод доступа к базе данных, называемый JDBC (на самом деле это название ничего не означает, хотя его часто расшифровывают как «Java Database Connectivity»):

```
/* Файл в Сети: Book.java */
import java.sql.*;

public class Book
{
    public static void main(String[] args) throws SQLException
    {
        // Инициализация драйвера с попыткой подключения к базе данных

        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());
        Connection conn =
            DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",
                                       "scott", "tiger");

        // Метод prepareCall использует "call"-синтаксис ANSI92
        CallableStatement cstmt = conn.prepareCall("{? = call booktitle(?)})");

        // Связывание переменных и параметров
        cstmt.registerOutParameter(1, Types.VARCHAR);
        cstmt.setString(2, "0-596-00180-0");

        // Теперь можно выполнить запрос и получить результат,
        // закрыть подключение и вывести результат.
        cstmt.executeUpdate();
        String bookTitle = cstmt.getString(1);
        conn.close();
        System.out.println(bookTitle);
    }
}
```

В этом примере используется «тонкий» драйвер, отличающийся прекрасной совместимостью и простотой установки (все необходимое для сетевых протоколов имеется в библиотеке Java) — но за счет производительности передачи данных. В качестве альтернативы можно использовать так называемый драйвер OCI. Пусть вас не пугает название — он совсем не так сложен, как одноименный интерфейс, упоминавшийся в предыдущем разделе.

Perl, с использованием Perl DBI и DBD::Oracle

Столь популярный в сообществе системных администраторов язык Perl можно назвать прародителем всех языков с открытым исходным кодом. Сейчас, в версии 5.10, он может практически все и распространен, кажется, повсеместно. А благодаря таким удобным средствам, как CPAN (Comprehensive Perl Archive Network — обширный сетевой архив

ресурсов для языка Perl), ничего не стоит установить интерфейсные модули типа DBI (DataBase Interface) и соответствующий драйвер Oracle, DBD::Oracle.

```
/* Файл в Сети: callbooktitle.pl */
#!/usr/bin/perl

use strict;
use DBI qw(:sql_types);

# Установить подключение или аварийно завершить приложение
my $dbh = DBI->connect(
    'dbi:Oracle:o92',
    'scott',
    'tiger',
    {
        RaiseError => 1,
        AutoCommit => 0
    }
) || die "Соединение с базой данных не установлено: $DBI::errstr";

my $retval;

# Вызов Oracle для разбора инструкции
eval {
    my $func = $dbh->prepare(q{
        BEGIN
            :retval := booktitle(isbn_in => :bind1);
        END;
    });
};

# Связывание параметров и выполнение процедуры
$func->bind_param(":bind1", "0-596-00180-0");
$func->bind_param_inout(":retval", \$retval, SQL_VARCHAR);
$func->execute;

};

if( $@ ) {
    warn "Хранимая процедура выполнена с ошибкой: $DBI::errstr\n";
    $dbh->rollback;
} else {
    print "Хранимая процедура вернула значение: $retval\n";
}

# Не забыть отключиться
$dbh->disconnect;
```

С другой стороны, код Perl часто оказывается совершенно неудобочитаемым. К тому же этот язык не отличается ни быстротой, ни компактностью кода, хотя компилируемые версии по крайней мере решают проблему быстрогодействия.

За дополнительной информацией о взаимодействии между Perl и Oracle обращайтесь к книге *Programming the Perl DBI* (автор Alligator Descartes и Tim Bunce). О языке Perl написано немало превосходных книг, не говоря уже о сетевых ресурсах *perl.com* (сайт O'Reilly), *perl.org* и *cpan.org*.

PHP, с использованием расширений Oracle

Многие разработчики, использующие бесплатный и невероятно популярный веб-сервер Apache, также являются завзятыми сторонниками бесплатного и невероятно популярного языка программирования PHP. Язык PHP, часто применяемый для построения динамических веб-страниц, также может использоваться для построения графических

приложений или выполнения программ командной строки. Как и следовало ожидать, Oracle поддерживается в PHP наряду со многими другими базами данных. Более того, компания Oracle объединила усилия с Zend для включения «официально одобренной» поддержки Oracle в PHP¹.

В следующем примере используется семейство функций PHP, объединенных общим названием OCI8. Пусть цифра «8» в названии вас не смущает; функции должны работать со всеми версиями Oracle, от Oracle7 до 11g.

```
/* Файл в Сети: callbooktitle.php */
<?PHP
// Создание подключения к базе данных o92
$conn = OCILogon ("scott", "tiger", "o92");

// Вызов Oracle для разбора инструкции
$stmt = OCIParse($conn,
    "begin :res := booktitle('0-596-00180-0'); end;");

// Вывод информации об ошибках
if (!$stmt) {
    $err = OCIError();
    echo "Oops, you broke it: ".$err["message"];
    exit;
}

// Привязка 200 символов переменной $result к заполнителю :res
OCIBindByName($stmt, "res", &$result, 200);

// Выполнение
OCIExecute($stmt);

// Сохранение значения в переменной
OCIResult($stmt,$result);

// Вывод в стандартный поток
echo "$result\n";

// Отключение от базы данных
OCILogoff($conn);
?>
```

При выполнении в командной строке результат выглядит примерно так:

```
$ php callbooktitle.php
Learning Oracle PL/SQL
```

Кстати говоря, функции Oracle OCI недоступны в PHP по умолчанию. Тем не менее ваш системный администратор без труда построит PHP с расширениями Oracle.

За дополнительной информацией обращайтесь по адресу <http://www.php.net> или к многочисленным книгам издательства O'Reilly по этой теме. Рекомендации по работе с Oracle из PHP приведены на сайте Oracle Technology Network.

PL/SQL Server Pages

Хотя технология PL/SQL Server Pages (PSP) запатентована Oracle, я решил упомянуть о ней, потому что она позволяет быстро создать работоспособную веб-страницу. Она также основана на прекомпиляции и позволяет вставлять код PL/SQL в HTML-страницы.

¹ Учтите, что по вопросам технической поддержки, относящейся к PHP, следует обращаться к сообществу пользователей или к таким компаниям, как Zend. Oracle не предоставляет поддержки PHP.

Конструкция `<%= %>` означает: «обработать как фрагмент PL/SQL и включить результат в страницу».

```
/* Файл в Сети: favorite_plsql_book.psp */
<%@ page language="PL/SQL" %>
<%@ plsql procedure="favorite_plsql_book" %>
<HTML>
  <HEAD>
    <TITLE>My favorite book about PL/SQL</TITLE>
  </HEAD>
  <BODY>
    <%= booktitle( '0-596-00180-0' ) %>
  </BODY>
</HTML>
```

Если приведенная страница будет правильно установлена на веб-сервере, подключенном к базе данных Oracle, то страница будет выглядеть так, как показано на рис. 2.3.



Рис. 2.3. Результат обработки PL/SQL Server Pages

Технология PL/SQL Server Pages пришлась по душе многим программистам и пользователям, так как она позволяет быстро и удобно создавать сайты.

За дополнительной информацией обращайтесь к книге *Learning Oracle PL/SQL*, написанной теми же авторами, что и книга, которую вы сейчас читаете.

Что же дальше?

Итак, вы познакомились с примерами использования PL/SQL в SQL*Plus и в других распространенных средах и языках программирования. Кроме того, код PL/SQL можно:

- встраивать в программы на языках COBOL и FORTRAN с последующей обработкой прекомпилятором Oracle;
- вызывать из языка Visual Basic с помощью разновидности ODBC;
- вызывать из языка программирования Ada с помощью технологии, называемой SQL*Module;
- выполнять автоматически по триггерам событий, происходящих в базе данных Oracle (например, при обновлении таблицы);
- включать в расписание для выполнения в базе данных Oracle с использованием встроенного пакета DBMS_SCHEDULER;
- использовать для манипуляций с содержимым TimesTen — технологии работы с базой данных в памяти, приобретенной Oracle Corporation. С ее содержимым можно работать из кода PL/SQL, как и с реляционными базами данных.

К сожалению, мы не сможем рассмотреть все эти возможности в последующих главах.

3

ОСНОВЫ ЯЗЫКА

Каждый язык, будь то естественный или компьютерный, имеет определенный синтаксис, лексикон и набор символов. Чтобы общаться на этом языке, необходимо изучить правила его использования. Многие с опаской приступают к изучению новых компьютерных языков, но обычно они очень просты, и PL/SQL не является исключением. Трудности общения на компьютерных языках связаны не с самим языком, а с компилятором или компьютером, с которым мы «общаемся». Компиляторы не обладают творческим, гибким мышлением, а их лексикон крайне ограничен. Разве что сообщают они очень, очень быстро... но только в рамках заданных правил.

Если приказать PL/SQL «подкинь-ка мне еще с полдюжины записей», едва ли вы получите требуемое. С точки зрения синтаксиса, для использования PL/SQL нужно расставлять все точки над «і». Поэтому в данной главе изложены основные правила языка, которые помогут вам общаться с компилятором PL/SQL — структура блоков PL/SQL, набор символов, лексические единицы и ключевое слово PRAGMA.

Структура блока PL/SQL

В PL/SQL, как и в большинстве других процедурных языков, наименьшей единицей группировки кода является *блок*. Он представляет собой фрагмент программного кода, определяющий границы выполнения и области видимости для объявлений переменных и обработки исключений. PL/SQL позволяет создавать как *именованные*, так и *анонимные блоки* (то есть блоки, не имеющие имени), которые представляют собой пакеты, процедуры, функции, триггеры или объектные типы.

Блок PL/SQL может содержать до четырех разделов, однако только один из них является обязательным.

- *Заголовок*. Используется только в именованных блоках, определяет способ вызова именованного блока или программы. Не обязателен.
- *Раздел объявлений*. Содержит описания переменных, курсоров и вложенных блоков, на которые имеются ссылки в исполняемом разделе и разделе исключений. Не обязателен.
- *Исполняемый раздел*. Команды, выполняемые ядром PL/SQL во время работы приложения. Обязателен.

- *Раздел исключений.* Обрабатывает исключения (предупреждения и ошибки). Не обязателен.

Структура блока PL/SQL для процедуры показана на рис. 3.1.

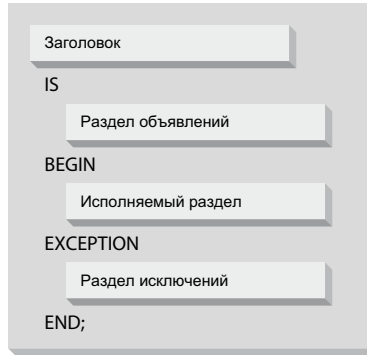


Рис. 3.1. Структура блока PL/SQL

На рис. 3.2 показана процедура, содержащая все четыре раздела. Этот конкретный блок начинается с ключевого слова `PROCEDURE` и, как и все блоки, завершается ключевым словом `END`.

```

PROCEDURE get_happy (ename_in IN VARCHAR2)  — Заголовок
IS
  l_hiredate DATE;                          — Раздел объявлений
BEGIN
  l_hiredate := SYSDATE - 2;
  INSERT INTO employee
    (emp_name, hiredate)
  VALUES (ename_in, l_hiredate);           — Исполняемый раздел
EXCEPTION
  WHEN DUP_VAL_IN_INDEX
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Cannot insert.');
```

END;

— Раздел исключений

Рис. 3.2. Процедура, содержащая все четыре раздела

Анонимные блоки

Когда кто-то хочет остаться неизвестным, он не называет своего имени. То же можно сказать и об анонимном блоке PL/SQL, показанном на рис. 3.3: в нем вообще нет раздела заголовка, блок начинается ключевым словом `DECLARE` (или `BEGIN`). Анонимный блок не может быть вызван из другого блока, поскольку он не имеет идентификатора, по которому к нему можно было бы обратиться. Таким образом, анонимный блок представляет собой контейнер для хранения команд PL/SQL — обычно с вызовами процедур и функций. Поскольку анонимные блоки могут содержать собственные разделы объявлений и исключений, разработчики часто используют вложение анонимных блоков для ограничения области видимости идентификаторов и организации обработки исключений в более крупных программах.


```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Hello world');
END;
```

● — Только исполняемый раздел

Рис. 3.3. Анонимный блок без разделов объявлений и исключений

Общий синтаксис анонимного блока PL/SQL:

```
[ DECLARE    ... объявления ... ]
BEGIN    ... одна или несколько исполняемых команд ...
[ EXCEPTION
    ... команды обработки исключений ... ]
END;
```

Квадратными скобками обозначаются необязательные составляющие синтаксиса. Анонимный блок обязательно содержит ключевые слова **BEGIN** и **END**, и между ними должна быть как минимум одна исполняемая команда. Несколько примеров:

- Простейший анонимный блок:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(SYSDATE);
END;
```

- Анонимный блок с добавлением раздела объявлений:

```
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE (l_right_now);
END;
```

- Тот же блок, но с разделом исключений:

```
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE (l_right_now);
EXCEPTION
  WHEN VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE('l_right_now не хватает места '
      || ' для стандартного формата даты');
END;
```

Анонимный блок выполняет серию команд, а затем завершает свою работу, то есть по сути является аналогом процедуры. Фактически каждый анонимный блок является анонимной процедурой. Они используются в различных ситуациях, в которых код PL/SQL выполняется либо непосредственно, либо как часть другой программы. Типичные примеры:

- *Триггеры баз данных.* Как упоминается в главе 19, триггеры выполняют анонимные блоки при наступлении определенных событий.
- *Специализированные команды или сценарии.* В SQL*Plus и других аналогичных средах анонимные блоки активизируются из кода, введенного вручную, или из сценариев, называемых хранимыми программами. Кроме того, команда SQL*Plus **EXECUTE** преобразует свой аргумент в анонимный блок, заключая его между ключевыми словами **BEGIN** и **END**.
- *Откомпилированная программа 3GL.* В Pro*C и OCI анонимные блоки используются для внедрения вызовов хранимых программ во внешний код.

Во всех случаях контекст — и возможно, механизм присваивания имени — предоставляется внешним объектом (будь то триггер, программа командной строки или откомпилированная программа).

Именованные блоки

Хотя анонимные блоки PL/SQL применяются во многих приложениях Oracle, вероятно, большая часть написанного вами кода будет оформлена в виде именованных блоков. Ранее вы уже видели несколько примеров хранимых процедур (см. рис. 3.1) и знаете, что их главной особенностью является наличие заголовка. Заголовок процедуры выглядит так:

```
PROCEDURE [схема.]имя [ ( параметр [, параметр ... ] ) ]
  [AUTHID {DEFINER | CURRENT_USER}]
```

Заголовок функции в целом очень похож на него, но дополнительно содержит ключевое слово RETURN:

```
FUNCTION [схема.]имя [ ( параметр [, параметр ... ] ) ]
  RETURN возвращаемый_тип
  [AUTHID {DEFINER | CURRENT_USER}]
  [DETERMINISTIC]
  [PARALLEL ENABLE ...]
  [PIPELINED [USING...] | AGGREGATE USING...]
```

Поскольку Oracle позволяет вызывать некоторые функции из SQL-команд, заголовок функции содержит больше необязательных компонентов, чем заголовок процедуры (в зависимости от функциональности и производительности исполнительный среды SQL).

Процедуры и функции более подробно рассматриваются в главе 17.

Вложенные блоки

PL/SQL, как и языки Ada и Pascal, относится к категории *языков с блочной структурой*, то есть блоки PL/SQL могут вкладываться в другие блоки. С другой стороны, язык C тоже поддерживает блоки, но стандартный C не является строго блочно-структурированным языком, потому что вложение подпрограмм в нем не допускается.

В следующем примере PL/SQL показана процедура, содержащая анонимный вложенный блок:

```
PROCEDURE calc_totals
IS
  year_total NUMBER;
BEGIN
  year_total := 0;

  /* Начало вложенного блока */
  DECLARE
    month_total NUMBER;
  BEGIN
    month_total := year_total / 12;
  END set_month_total;
  /* Конец вложенного блока */

END;
```

Ограничители */** и **/* обозначают начало и конец комментариев (см. раздел «Комментарии» далее в этой главе). Анонимные блоки также могут вкладываться более чем на один уровень (рис. 3.4).

```

DECLARE
  CURSOR emp_cur IS ...;
BEGIN
  DECLARE
    total_sales NUMBER;
  BEGIN
    DECLARE
      l_hiredate DATE;
    BEGIN
      ...
    END;
  END;
END;

```

Рис. 3.4. Вложенные анонимные блоки

Главное преимущество вложенных блоков заключается в том, что они позволяют ограничивать области видимости и действия синтаксических элементов кода.

Область действия

В любом языке программирования *областью действия* (scope) называется механизм определения «сущности», обозначаемой некоторым идентификатором. Если программа содержит более одного экземпляра идентификатора, то используемый экземпляр определяется языковыми правилами области действия. Управление областью видимости идентификаторов не только помогает контролировать поведение программы, но и уменьшает вероятность того, что программист по ошибке изменит значение не той переменной.

В PL/SQL переменные, исключения, модули и некоторые другие структуры являются локальными для блока, в котором они объявлены. Когда выполнение блока будет завершено, все эти структуры становятся недоступными. Например, в приведенной выше процедуре `calc_totals` можно обращаться к элементам внешнего блока (например, к переменной `year_total`), тогда как элементы, объявленные во внутреннем блоке, для внешнего блока недоступны.

У каждой переменной PL/SQL имеется некоторая область действия — участок программы (блок, подпрограмма или пакет), в котором можно сослаться на эту переменную. Рассмотрим следующее определение пакета:

```

PACKAGE scope_demo
IS
  g_global  NUMBER;

  PROCEDURE set_global (number_in IN NUMBER);
END scope_demo;

PACKAGE BODY scope_demo
IS
  PROCEDURE set_global (number_in IN NUMBER)
  IS
    l_salary  NUMBER := 10000;
    l_count   PLS_INTEGER;
  BEGIN

    <<local_block>>
    DECLARE
      l_inner  NUMBER;
    BEGIN
      SELECT COUNT (*)
      INTO l_count

```

продолжение ➤

```

        FROM employees
        WHERE department_id = l_inner AND salary > l_salary;
    END local_block;

    g_global := number_in;
END set_global;
END scope_demo;

```

Переменная `scope_demo.g_global` может использоваться в любом блоке любой схемы, обладающем привилегией `EXECUTE` для `scope_demo`.

Переменная `l_salary` может использоваться только в процедуре `set_global`.

Переменная `l_inner` может использоваться только в локальном или вложенном блоке; обратите внимание на использование метки `local_block` для присваивания имени вложенному блоку.

Уточнение ссылок на переменные и столбцы в командах SQL

Ссылки на переменные и столбцы в предыдущем примере *не уточнялись* именами области действия. Далее приводится другая версия того же пакета, но на этот раз с уточнением ссылок (выделены полужирным шрифтом):

```

PACKAGE BODY scope_demo
IS
    PROCEDURE set_global (number_in IN NUMBER)
    IS
        l_salary  NUMBER := 10000;
        l_count   PLS_INTEGER;
    BEGIN
        <<local_block>>
        DECLARE
            l_inner  PLS_INTEGER;
        BEGIN
            SELECT COUNT (*)
            INTO set_global.l_count
            FROM employees e
            WHERE e.department_id = local_block.l_inner
            AND e.salary > set_global.l_salary;
        END local_block;
        scope_demo.g_global := set_global.number_in;
    END set_global;
END scope_demo;

```

В новой версии каждая ссылка на столбец и переменную уточняется псевдонимом таблицы, именем пакета, именем процедуры или меткой вложенного блока.

Итак, теперь вы знаете об этой возможности — но зачем тратить время на уточнение имен? Для этого есть несколько очень веских причин:

- Удобство чтения кода.
- Предотвращение ошибок, возникающих при совпадении имен переменных с именами столбцов.
- Возможность использования детализированных зависимостей, появившаяся в Oracle11g и подробно рассматриваемая в главе 20.

Давайте поближе рассмотрим первые две из этих причин. Третья будет описана в главе 20.

Удобство чтения

Практически любая команда SQL, встроенная в программу PL/SQL, содержит ссылки на столбцы и переменные. В небольших, простых командах SQL различать эти ссылки относительно просто. Однако во многих приложениях используются очень длинные, исключительно сложные команды SQL с десятками и даже сотнями ссылок на столбцы и переменные.

Без уточнения ссылок вам будет намного сложнее различать переменные и столбцы. С уточнениями сразу видно, к чему относится та или иная ссылка.

«Один момент... Мы используем четко определенные схемы назначения имен, при помощи которых мы различаем строки и столбцы. Имена всех локальных переменных начинаются с „l_“, поэтому мы сразу видим, что идентификатор представляет локальную переменную».

Да, все правильно; все мы должны иметь (и соблюдать) правила назначения имен, чтобы имена идентификаторов содержали дополнительную информацию о них (что это — параметр, переменная? К какому типу данных она относится?).

Безусловно, правила назначения имен полезны, но они еще *не гарантируют*, что компилятор PL/SQL всегда будет интерпретировать ваши идентификаторы именно так, как вы задумали.

Предотвращение ошибок

Если не уточнять ссылки на переменные PL/SQL во встроенных командах SQL, код, который правильно работает сегодня, может внезапно утратить работоспособность в будущем. И разработчику будет очень трудно понять, что же пошло не так.

Вернемся к встроенной команде SQL без уточнения ссылок:

```
SELECT COUNT (*)  
  INTO l_count  
  FROM employees  
 WHERE department_id = l_inner AND salary > l_salary;
```

Сегодня идентификатор `l_salary` однозначно представляет переменную `l_salary`, объявленную в процедуре `set_global`. Я тестирую свою программу — она работает! Программа поставляется клиентам, все довольны.

А через два года пользователи просят своего администратора базы данных добавить в таблицу `employees` столбец, которому по случайности присваивается имя «`l_salary`». Видите проблему?

Во встроенной команде SQL база данных Oracle всегда начинает поиск соответствия для неуточненных идентификаторов со столбцов таблиц. Если найти столбец с указанным именем не удалось, Oracle переходит к поиску среди переменных PL/SQL в области действия. После добавления в таблицу `employee` столбца `l_salary` моей неуточненной ссылке `l_salary` в команде `SELECT` ставится в соответствие не переменная PL/SQL, а столбец таблицы. Результат?

Пакет `scope_demo` по-прежнему компилируется без ошибок, но секция `WHERE` запроса ведет себя не так, как ожидалось. База данных не использует значение переменной `l_salary`, а сравнивает значение столбца `salary` в строке таблицы `employees` со значением столбца `l_salary` той же строки. Отыскать подобную ошибку бывает очень непросто!

Не полагайтесь *только* на правила назначения имен для предотвращения «коллизий» между идентификаторами; уточняйте ссылки на все имена столбцов и переменных во встроенных командах SQL. Это существенно снизит риск непредсказуемого поведения программ в будущем при возможных модификациях таблиц.

Видимость

Важным свойством переменной, связанным с областью ее действия, является видимость. Данное свойство определяет, можно ли обращаться к переменной только по ее имени, или же к имени необходимо добавлять префикс.

«Видимые» идентификаторы

Начнем с тривиального случая:

```
DECLARE
    first_day DATE;
    last_day DATE;
BEGIN
    first_day := SYSDATE;
    last_day := ADD_MONTHS (first_day, 6);
END;
```

Обе переменные `first_day` и `last_day` объявляются в том же блоке, где они используются, поэтому при обращении к ним указаны только имена без уточняющих префиксов. Такие идентификаторы называются *видимыми*. В общем случае видимым идентификатором может быть:

- идентификатор, объявленный в текущем блоке;
- идентификатор, объявленный в блоке, который включает текущий блок;
- отдельный объект базы данных (таблица, представление и т. д.) или объект PL/SQL (процедура, функция), владельцем которого вы являетесь;
- отдельный объект базы данных или объект PL/SQL, на который у вас имеются соответствующие привилегии и который определяется видимым синонимом;
- индексная переменная цикла (видима и доступна только внутри цикла).

PL/SQL также позволяет обращаться к существующим объектам, которые не находятся в пределах непосредственной видимости блока. О том, как это делается, рассказано в следующем разделе.

Уточненные идентификаторы

Типичным примером идентификаторов, невидимых в области кода, где они используются, являются идентификаторы, объявленные в спецификации пакета (имена переменных, типы данных, имена процедур и функций). Чтобы обратиться к такому объекту, необходимо указать перед его именем префикс и точку (аналогичным образом имя столбца уточняется именем таблицы, в которой он содержится). Например:

- `price_util.compute_means` — программа с именем `compute_means` из пакета `price_util`.
- `math.pi` — константа с именем `pi`, объявленная и инициализированная в пакете `math`.

Дополнительное уточнение может определять владельца объекта. Например, выражение `scott.price_util.compute_means`

обозначает процедуру `compute_means` пакета `price_util`, принадлежащего пользователю Oracle с учетной записью `scott`.

Уточнение идентификаторов именами модулей

PL/SQL предоставляет несколько способов уточнения идентификаторов для логического разрешения ссылок. Так, использование пакетов позволяет создавать переменные с глобальной областью действия. Допустим, имеется пакет `company_pkg` и в спецификации пакета объявлена переменная с именем `last_company_id`:

```
PACKAGE company_pkg
IS
    last_company_id NUMBER;
    ...
END company_pkg;
```

На переменную можно ссылаться за пределами пакета — необходимо лишь указать перед ее именем имя пакета:

```
IF new_company_id = company_pkg.last_company_id THEN
```

По умолчанию значение, присвоенное переменной пакетного уровня, продолжает действовать на протяжении текущего сеанса базы данных; оно не выходит из области действия вплоть до разрыва подключения.

Идентификатор также можно уточнить именем модуля, в котором он определен:

```
PROCEDURE calc_totals
IS
    salary NUMBER;
BEGIN
    ...
    DECLARE
        salary NUMBER;
    BEGIN
        salary := calc_totals.salary;
    END;
    ...
END;
```

В первом объявлении создается переменная **salary**, областью действия которой является вся процедура. Однако затем во вложенном блоке объявляется другой идентификатор с тем же именем. Поэтому ссылка на переменную **salary** во внутреннем блоке всегда сначала разрешается по объявлению в этом блоке, где переменная видима безо всяких уточнений. Чтобы во внутреннем блоке обратиться к переменной **salary**, объявленной на уровне процедуры, необходимо уточнить ее имя именем процедуры (**calc_totals.salary**). Этот метод уточнения идентификаторов работает и в других контекстах. Что произойдет при выполнении следующей процедуры (**order_id** — первичный ключ таблицы **orders**):

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
    DELETE orders WHERE order_id = order_id; -- Катастрофа!
END;
```

Этот фрагмент удалит из таблицы **orders** все записи независимо от переданного значения **order_id**. Дело в том, что механизм разрешения имен SQL сначала проверяет имена столбцов и только потом переходит к идентификаторам PL/SQL. Условие **WHERE (order_id = order_id)** всегда истинно, поэтому все данные пропадают.

Возможное решение проблемы выглядит так:

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
    DELETE orders WHERE order_id = remove_order.order_id;
END;
```

В этом случае при разборе имя переменной будет интерпретировано правильно. (Решение работает даже при наличии в пакете функции с именем **remove_order.order_id**.)

В PL/SQL установлен целый ряд правил разрешения конфликтов имен, а этой проблеме уделяется серьезное внимание. И хотя знать эти правила полезно, лучше использовать уникальные идентификаторы, чтобы избежать подобных конфликтов. Старайтесь писать

надежный код! Если же вы не хотите уточнять каждую переменную, чтобы обеспечить ее уникальность, вам придется тщательно проработать схему назначения имен для предотвращения подобных конфликтов.

Вложенные программы

Завершая тему вложения, области действия и видимости, стоит упомянуть о такой полезной возможности PL/SQL, как *вложенные программы* (nested programs). Вложенная программа представляет собой процедуру или функцию, которая полностью размещается в разделе объявлений внешнего блока. Вложенная программа может обращаться ко всем переменным и параметрам, объявленным ранее во внешнем блоке, как показывает следующий пример:

```
PROCEDURE calc_totals (fudge_factor_in IN NUMBER)
IS
    subtotal NUMBER := 0;

    /* Начало вложенного блока (в данном случае процедуры).
    | Обратите внимание: процедура полностью размещается
    | в разделе объявлений calc_totals.
    */
    PROCEDURE compute_running_total (increment_in IN PLS_INTEGER)
    IS
    BEGIN
        /* Переменная subtotal (см. выше) видима и находится в области действия */
        subtotal := subtotal + increment_in * fudge_factor_in;
    END;
    /* Конец вложенного блока */
BEGIN
    FOR month_idx IN 1..12
    LOOP
        compute_running_total (month_idx);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Годовой итог: ' || subtotal);
END;
```

Вложенные программы упрощают чтение и сопровождение кода, а также позволяют повторно использовать логику, задействованную в нескольких местах блока. За дополнительной информацией по этой теме обращайтесь к главе 17.

Набор символов PL/SQL

Программа PL/SQL представляет собой последовательность команд, состоящих из одной или нескольких строк текста. Набор символов, из которых составляются эти строки, зависит от используемого в базе данных набора символов. Для примера в табл. 3.1 приведены символы набора US7ASCII.

Таблица 3.1. Символы набора US7ASCII

| Тип | Символы |
|-----------------------|--|
| Буквы | A-Z, a-z |
| Цифры | 0-9 |
| Знаки | ~ ! @ # \$ % * () _ — + = : ; " ' < > , . ? / ^ |
| Пропуски (whitespace) | Табуляция, пробел, новая строка, возврат каретки |

Каждое ключевое слово, оператор и лексема PL/SQL состоит из разных комбинаций символов данного набора. Остается разобраться, как связать эти символы друг с другом!

А теперь любопытный факт о PL/SQL. В документации Oracle (а также в предыдущих изданиях этой книги) символы `&`, `{ }` и `[]` также были отнесены к стандартному набору символов. Хотя эти символы допустимы в литеральных строках, Oracle не использует их в коде PL/SQL. Более того, эти символы не могут непосредственно включаться в идентификаторы, определяемые программистом.

Запомните, что язык PL/SQL *не учитывает регистр символов*. Это означает, что символы верхнего регистра интерпретируются так же, как символы нижнего регистра (кроме символов, заключенных в ограничители, с которыми они интерпретируются как литеральная строка). В книге встроенные ключевые слова языка (а также некоторые идентификаторы, используемые Oracle в качестве имен встроенных функций и пакетов) обычно записываются в верхнем регистре, а идентификаторы, определяемые программистом, — в нижнем регистре.

Некоторые символы — как по отдельности, так и в сочетании с другими символами — имеют в PL/SQL специальное значение. Эти специальные символы (простые и составные) перечислены в табл. 3.2.

Таблица 3.2. Простые и составные специальные символы в PL/SQL

| Символы | Описание |
|---|---|
| <code>;</code> | Завершает объявления и команды |
| <code>%</code> | Индикатор атрибута (атрибут курсора, подобный <code>%ISOPEN</code> , или атрибут неявных объявлений, например <code>%ROWTYPE</code>) также используется в качестве символа подстановки в условии <code>LIKE</code> |
| <code>_</code> | Обозначение подстановки одного символа в условии <code>LIKE</code> |
| <code>@</code> | Признак удаленного местоположения |
| <code>:</code> | Признак хост-переменной, например <code>:block.item</code> в Oracle Forms |
| <code>**</code> | Оператор возведения в степень |
| <code><></code> или <code>!=</code> или <code>^=</code> или <code>~=</code> | Оператор сравнения «не равно» |
| <code> </code> | Оператор конкатенации |
| <code><< и >></code> | Ограничители метки |
| <code><= и >=</code> | Операторы сравнения «меньше или равно» и «больше или равно» |
| <code>:=</code> | Оператор присваивания |
| <code>=></code> | Оператор ассоциации |
| <code>..</code> | Оператор диапазона |
| <code>--</code> | Признак однострочного комментария |
| <code>/* и */</code> | Начальный и конечный ограничители многострочного комментария |

Символы группируются в лексические единицы, которые называются *атомарными*, поскольку они представляют собой наименьшие самостоятельные элементы языка. В PL/SQL лексическими единицами являются идентификатор, литерал, разделитель и комментарий. Мы рассмотрим их в следующих разделах.

Идентификатор

Идентификатор — это имя объекта данных PL/SQL, которым может быть:

- константа или переменная;
- исключение;
- курсор;
- имя программы: процедура, функция, пакет, объектный тип, триггер и т. д.;
- зарезервированное слово;
- метка.

Идентификаторы PL/SQL обладают следующими свойствами:

- длина — до 30 символов;
- должны начинаться с буквы;
- могут включать символы «\$», «_» и «#»;
- не должны содержать пропусков.

Если два идентификатора различаются только регистром одного или нескольких символов, PL/SQL обычно воспринимает их как один идентификатор¹. Например, следующие идентификаторы считаются одинаковыми:

```
lots_of_$MONEY$
LOTS_of_$MONEY$
lots_of_$Money$
```

Примеры допустимых имен идентификаторов:

```
company_id#
primary_acct_responsibility
First_Name
FirstName
address_line1
S123456
```

Идентификаторы, приведенные ниже, в PL/SQL недопустимы:

```
1st_year           -- Не начинается с буквы
procedure-name     -- Содержит недопустимый символ "-"
minimum_%_due      -- Содержит недопустимый символ "%"
maximum_value_exploded_for_detail -- Имя слишком длинное
company ID         -- Имя не может содержать пробелов
```

Идентификаторы используются для обращения к элементам программы и являются одним из основных средств передачи информации другим программистам. По этой причине во многих организациях принимаются стандартные правила выбора имен; даже если в вашем проекте такие правила отсутствуют, имена все равно должны быть содержательными... даже если вы будете единственным человеком, который когда-либо увидит код!

Некоторые из правил именования объектов можно нарушить, заключая идентификатор в кавычки. Мы не рекомендуем пользоваться этим методом, но когда-нибудь вам может встретиться хитроумный код вроде следующего:

```
SQL> DECLARE
2   "pi" CONSTANT NUMBER := 3.141592654;
3   "PI" CONSTANT NUMBER := 3.14159265358979323846;
4   "2 pi" CONSTANT NUMBER := 2 * "pi";
5 BEGIN
6   DBMS_OUTPUT.PUT_LINE('pi: ' || "pi");
7   DBMS_OUTPUT.PUT_LINE('PI: ' || pi);
8   DBMS_OUTPUT.PUT_LINE('2 pi: ' || "2 pi");
9 END;
10 /

pi: 3.141592654
PI: 3.14159265358979323846
2 pi: 6.283185308
```

Обратите внимание: в строке 7 идентификатор `pi` используется без кавычек. Поскольку компилятор преобразует все идентификаторы и ключевые слова в верхний регистр, данный идентификатор относится к переменной, объявленной в строке 3 под именем `PI`.

¹ Для этого компилятор на ранней стадии компиляции преобразует текст программы в верхний регистр.

Иногда прием с кавычками применяется в SQL-инструкциях для ссылки на таблицы базы данных, имена столбцов которых содержат символы разных регистров (например, если программист использовал Microsoft Access для создания таблиц Oracle).

Зарезервированные слова

Конечно, выбор идентификаторов в программе нельзя назвать полностью произвольным. Некоторые идентификаторы (такие, как `BEGIN`, `IF` и `THEN`) имеют в языке PL/SQL специальное значение.

В PL/SQL встроенные идентификаторы делятся на два вида:

- ключевые слова;
- идентификаторы пакета `STANDARD`.

Вы не должны (а в большинстве случаев и *не сможете*) использовать их в качестве имен объектов, объявляемых в своих программах.

Ключевые слова

Некоторые идентификаторы с точки зрения компилятора PL/SQL имеют строго определенную семантику. Иначе говоря, вы не сможете определить переменную с именем идентификатора. К таковым относится, например, слово `END`, завершающее программу, условная команда `IF` и команды цикла. При попытке объявить переменную с именем `end`:

```
DECLARE
  end VARCHAR2(10) := 'blip';  /* не работает; "end" нельзя использовать
                                в качестве имени переменной. */
BEGIN
  DBMS_OUTPUT.PUT_LINE (end);
END;
/
```

компилятор выдает сообщение об ошибке:

```
PLS-00103: Encountered the symbol "END" when expecting one of the following:
...
```

Идентификаторы пакета `STANDARD`

Также не следует использовать в качестве идентификаторов имена объектов, определенные в специальном встроенном пакете `STANDARD` — одном из двух стандартных пакетов PL/SQL. В нем объявлено большое количество основных элементов языка PL/SQL, включая типы данных (например, `PLS_INTEGER`) идентификаторов — имена встроенных исключений (`DUP_VAL_ON_INDEX` и т. д.), функций (например, `UPPER`, `REPLACE` и `TO_DATE`).

Вопреки распространенному мнению, идентификаторы пакета `STANDARD` (и `DBMS_STANDARD`, другого стандартного пакета) *не являются* ключевыми словами. Вы *можете* объявлять собственные переменные с такими же именами, и программа успешно откомпилируется. Тем не менее это создаст изрядную путаницу в вашем коде.

Пакет `STANDARD` подробно рассматривается в главе 24.

Как избежать использования зарезервированных слов

Поиск допустимого имени для идентификатора — далеко не самая серьезная проблема, так как существуют многие тысячи комбинаций допустимых символов. Вопрос в другом: как узнать, не используете ли вы зарезервированное слово в своей программе? Прежде всего компилятор сообщит о попытке использования зарезервированного слова в качестве идентификатора. Если ваша любознательность этим не ограничивается, постройте

запрос к представлению V\$RESERVED_WORDS и попробуйте откомпилировать динамически построенный блок PL/SQL, использующий зарезервированное слово в качестве идентификатора. Я так и поступил; соответствующий сценарий хранится в файле `reserved_words.sql` на сайте книги. Выходные данные сценария находятся в файле `reserved.txt`.

Результаты очень интересны. Общая сводка выглядит так:

```
Reserved Word Analysis Summary
Total count in V$RESERVED_WORDS = 1733
Total number of reserved words = 118
Total number of non-reserved words = 1615
```

Итак, подавляющее большинство слов, которые Oracle включает в это представление, не являются формально зарезервированными; эти слова могут использоваться в качестве имен идентификаторов.

В общем случае я рекомендую избегать тех слов, которые Oracle Corporation использует в своих технологиях. А еще лучше — используйте правила назначения имен, основанные на последовательном применении префиксов и суффиксов. Такие схемы практически исключают случайное использование зарезервированных слов PL/SQL.

Пропуски и ключевые слова

Идентификаторы необходимо отделять друг от друга хотя бы одним пробелом или разделителем. Вы можете форматировать текст программы, вставляя дополнительные пробелы, разрывы строк и табуляции во всех местах, где могут находиться пробелы, — смысл кода при этом не изменится.

Например, две приведенные ниже команды эквивалентны:

```
IF too_many_orders
THEN
    warn_user;
ELSIF no_orders_entered
THEN
    prompt_for_orders;
END IF;

IF too_many_orders THEN warn_user;
ELSIF no_orders_entered THEN prompt_for_orders;
END IF;
```

С другой стороны, пробелы, табуляции и разрывы строк недопустимы внутри лексических единиц — таких, как оператор «не равно» (`!=`). Следующая команда приводит к ошибке компиляции:

```
IF max_salary != min_salary THEN    -- ошибка компиляции PLS-00103
потому что символы ! и = разделены пробелом.
```

Литералы

Литералом (literal) называется значение, с которым не связан идентификатор; оно существует «само по себе». Несколько примеров литералов, которые *могут* встретиться в программе PL/SQL:

○ Числа:

```
415, 21.6, 3.141592654f, 7D, NULL
```

○ Строки:

```
'This is my sentence', '01-OCT-1986', q'!hello!', NULL
```

○ Временные интервалы:

```
INTERVAL '25-6' YEAR TO MONTH, INTERVAL '-18' MONTH, NULL
```

○ Логические значения:

```
TRUE, FALSE, NULL
```

Суффикс «f» в числовом литерале 3.14159f обозначает 32-разрядное вещественное число в стандарте IEEE 754, частично поддерживаемом Oracle, начиная с версии 10g Release 1. Аналогичным образом, 7D — число 7, представленное в 64-разрядном вещественном формате.

О строке `q'!hello!'` следует упомянуть особо. Знак `!` — пользовательский ограничитель (возможность использования пользовательских ограничителей также появилась в Oracle10g). Начальная буква `q` и одинарные кавычки, в которые заключено значение, сообщают компилятору, что `!` — ограничитель, а не часть значения, а строка содержит просто слово `hello`.

Тип данных `INTERVAL` предназначен для работы с интервалами между определенными датами или моментами времени. Первый интервал в приведенном выше примере означает «через 25 лет и еще 6 месяцев», а второй — «на 18 месяцев ранее».

Хотя интервалы могут задаваться в литеральном формате, для типов `DATE` такая возможность не предусмотрена; скажем, значение `'01-OCT-1986'` интерпретируется как строка, а не как тип Oracle `DATE`. Да, PL/SQL или SQL *могут* неявно преобразовать строку `'01-OCT-1986'` во внутренний формат даты Oracle¹, но обычно в программном коде следует выполнять явные преобразования с использованием встроенных функций. Пример:

```
TO_DATE('01-OCT-1986', 'DD-MON-YYYY')  
TO_TIMESTAMP_TZ('01-OCT-1986 00:00:00 -6', 'DD-MON-YYYY HH24:MI:SS TZH')
```

Оба выражения возвращают время 00:00:00 1 октября 1986 года; первое относится к типу данных `DATE`, а второе — к типу данных временного штампа с часовым поясом. Во второе выражение также включена информация о часовом поясе (–6 — смещение в часах от общемирового времени GMT (UCT)).

В отличие от идентификаторов строковые литералы PL/SQL чувствительны к регистру символов. Как и следовало ожидать, следующие два литерала различаются:

```
'Steven'  
'steven'
```

А проверка следующего условия возвращает `FALSE`:

```
IF 'Steven' = 'steven'
```

NULL

Отсутствие значения представляется в Oracle ключевым словом `NULL`. Как показано в предыдущем разделе, переменные почти всех типов данных PL/SQL могут существовать в состоянии `NULL` (исключение составляют только ассоциативные массивы, экземпляры которых всегда отличны от `NULL`). Правильная обработка переменных со значением `NULL` и так является непростой задачей для программиста, а `NULL`-строки вдобавок требуют особого обращения.

¹ При условии, что параметру `NLS_DATE_FORMAT` базы данных или сеанса задано значение `DD-MON-YYYY`.

В Oracle SQL и PL/SQL NULL-строка *обычно* неотличима от литерала из нуля символов, буквально представляемого в виде '' (две одинарные кавычки, между которыми нет ни одного символа). Например, следующее условие равно TRUE как в SQL, так и в PL/SQL:

```
'' IS NULL
```

Если в PL/SQL переменной VARCHAR2(n) присваивается строка нулевой длины, то результат также считается равным NULL:

```
DECLARE
    str VARCHAR2(1) := '';
BEGIN
    IF str IS NULL    -- равно TRUE
```

Такое поведение соответствует правилам обработки столбцов VARCHAR2 в таблицах баз данных.

Но с данными типа CHAR дело обстоит сложнее. Если создать в PL/SQL переменную типа CHAR(n) и присвоить ей строку нулевой длины, база данных *дополняет пустую переменную пробелами*, в результате чего она становится отличной от NULL:

```
DECLARE
    flag CHAR(2) := ''; -- CHAR(2) присваивается строка нулевой длины
BEGIN
    IF flag = ' '    ...    -- равно TRUE
    IF flag IS NULL  ...    -- равно FALSE
```

Как ни странно, такое поведение проявляется только в PL/SQL. В базе данных при вставке строки нулевой длины в поле типа CHAR(n) содержимое столбца не дополняется пробелами, а остается равным NULL!

В этих примерах проявляется частичная поддержка Oracle версий 92 и 99 стандарта ANSI SQL, согласно которому строка нулевой длины должна отличаться от NULL-строки. По утверждениям Oracle, стандарт будет полностью поддерживаться в будущих версиях. Впрочем, разговоры на эту тему идут уже 15 лет и пока этого не произошло.

База данных пытается неявно преобразовать NULL к типу значения, необходимому для текущей операции. Время от времени может возникнуть необходимость в выполнении явных преобразований с использованием синтаксических конструкций вида TO_NUMBER(NULL) или CAST(NULL AS NUMBER).

Одинарные кавычки внутри строки

Неприятная сторона работы со строковыми литералами проявляется тогда, когда вам нужно включить в строку одинарную кавычку. До выхода Oracle10g одна литеральная кавычка внутри строки обозначалась удвоением символа кавычки. Примеры:

| Литерал (с ограничителем по умолчанию) | Фактическое значение |
|---|---|
| 'There's no business like show business.' | There's no business like show business. |
| ""Hound of the Baskervilles"" | "Hound of the Baskervilles" |
| '''' | ' |
| '''hello''' | 'hello' |
| '''''' | " |

Например, из этих примеров следует, что для представления литерала из двух одиночных кавычек требуется целых шесть символов. Для упрощения подобных конструкций в Oracle10g появились специальные ограничители. Литерал начинается с символа q, а выражение в ограничителях заключается в одиночные кавычки. В следующей таблице приведены примеры использования этого синтаксиса.

| Литерал (со специальными ограничителями) | Фактическое значение |
|---|---|
| q' (There's no business like show business.) ' | There's no business like show business. |
| q' { "Hound of the Baskervilles" } ' | "Hound of the Baskervilles" |
| q' [' '] ' | ' |
| q' !'hello' ! ' | 'hello' |
| q' ' ' ' | " |

Как видно из примеров, вы можете использовать как простые ограничители (!, | и т. д.), так и «парные»: левая и правая круглые, фигурные и квадратные скобки.

Напоследок следует заметить, что символ двойной кавычки в строковых литералах не имеет специального значения, а интерпретируется так же, как буква или цифра.

Числовые литералы

Числовые литералы могут быть целыми или действительными (то есть содержащими дробную часть) числами. Заметьте, что PL/SQL рассматривает число 154.00 как действительное, хотя его дробная часть равна нулю и с точки зрения математики оно является целым. Целые и действительные числа имеют разное внутреннее представление, и преобразование числа из одной формы в другую требует определенных ресурсов.

Для определения числовых литералов может использоваться экспоненциальная (научная) запись. Буква E (в верхнем или нижнем регистре) обозначает степень 10 (например, 3.05E19, 12e–5).

Начиная с версии Oracle Database 10g, вещественные числа могут представляться как типом Oracle NUMBER, так и стандартным типом IEEE 754 с плавающей точкой. К последней категории относятся типы BINARY (32-разрядное; обозначается суффиксом F) и BINARY DOUBLE (64-разрядное; обозначается суффиксом D).

В соответствии со стандартом IEEE, в некоторых выражениях могут использоваться именованные константы из следующей таблицы.

| Описание | BINARY (32-разрядное) | BINARY DOUBLE (64-разрядное) |
|--|----------------------------|------------------------------|
| «Не число» (NaN, Not a Number): результат деления на 0 или недействительной операции | BINARY_FLOAT_NAN | BINARY_DOUBLE_NAN |
| Положительная бесконечность | BINARY_FLOAT_INFINITY | BINARY_DOUBLE_INFINITY |
| Абсолютное наибольшее число, которое может быть представлено | BINARY_FLOAT_MAX_NORMAL | BINARY_DOUBLE_MAX_NORMAL |
| Наименьшее нормальное число; порог потери значимости | BINARY_FLOAT_MIN_NORMAL | BINARY_DOUBLE_MIN_NORMAL |
| Наибольшее положительное число, меньшее порога потери значимости | BINARY_FLOAT_MAX_SUBNORMAL | BINARY_DOUBLE_MAX_SUBNORMAL |
| Абсолютное наименьшее число, которое может быть представлено | BINARY_FLOAT_MIN_SUBNORMAL | BINARY_DOUBLE_MIN_SUBNORMAL |

Логические (булевские) литералы

Для представления логических значений в PL/SQL определено два литерала: TRUE и FALSE. Это не строки и их не нужно заключать в кавычки. Они используются для присваивания значений логическим переменным, как в следующем примере:

```
DECLARE
    enough_money BOOLEAN; -- Объявление логической переменной
BEGIN
    enough_money := FALSE; -- Присваивание значения
END;
```

При проверке логического значения литерал можно не указывать — тип переменной говорит сам за себя:

```
DECLARE
    enough_money BOOLEAN;
BEGIN
    IF enough_money
    THEN
        ...
```

Логическое выражение, переменная или константа также могут быть равны NULL — неопределенному значению, которое не является ни TRUE, ни FALSE. За дополнительной информацией обращайтесь к главе 4.

Точка с запятой как разделитель

Программа на PL/SQL представляет собой последовательность объявлений и команд, которые определяются логически, а не физически — иначе говоря, их границы определяются не физическим завершением строки кода, а специальным завершителем — символом точки с запятой (;). Более того, одна команда нередко распространяется на несколько строк для удобства чтения. Например, следующая команда IF занимает четыре строки, а отступы более наглядно выделяют логику ее работы:

```
IF salary < min_salary (2003)
THEN
    salary := salary + salary * .25;
END IF;
```

В ней присутствуют два символа точки с запятой. Первый отмечает конец единственной команды присваивания в конструкции IF-END IF, а второй — конец команды IF. Эту команду можно было бы записать в одной физической строке, результат будет одинаковым:

```
IF salary < min_salary (2003) THEN salary := salary + salary*.25; END IF;
```

Каждая исполняемая команда должна завершаться точкой с запятой, даже если она вложена в другую команду. Но если вы стремитесь к тому, чтобы ваш код нормально читался, мы не рекомендуем объединять разные компоненты команды IF в одной строке. И вообще каждую команду или объявление желательно размещать в отдельной строке.

```
IF salary < min_salary (2003)
THEN
    salary := salary + salary * .25;
END IF;
```

В нем содержится два символа точки с запятой. Первый отмечает конец единственной команды присваивания в конструкции IF-END IF, а второй — конец команды IF. Эту команду можно было бы записать в одной физической строке, результат будет одинаковым:

```
IF salary < min_salary (2003) THEN salary := salary + salary*.25; END IF;
```

Каждая исполняемая команда должна завершаться точкой с запятой, даже если она вложена в другую команду. Но если вы стремитесь к тому, чтобы ваш код *нормально* читался, мы не рекомендуем объединять разные компоненты команды IF в одной строке. И вообще каждую команду или объявление желательно размещать в отдельной строке.

Комментарии

Наличие поясняющего текста (*комментариев*) является важным признаком хорошей программы. В данной книге приводится множество советов, поясняющих, как сделать программы самодокументируемыми за счет использования продуманных соглашений об именах и модульного подхода. Однако для понимания сложного программного кода этого еще не достаточно. PL/SQL предлагает разработчикам две разновидности комментариев: однострочные и многострочные.

Однострочные комментарии

Однострочные комментарии начинаются с двух дефисов (--), между которыми не может быть пробелов или каких-либо других символов. Весь текст после двух дефисов и до конца физической строки рассматривается как комментарий и игнорируется компилятором. Если два дефиса стоят в начале строки, то комментарием считается вся строка.

Запомните: два дефиса помечают как комментарий только остаток физической строки, а не логической команды PL/SQL. В следующей команде IF однострочный комментарий поясняет логику условного выражения:

```
IF salary < min_salary (2003) -- Функция возвращает минимальную годовую зарплату.  
THEN  
    salary := salary + salary*.25;  
END IF;
```

Многострочные комментарии

Если однострочные комментарии удобны для краткого описания фрагментов кода или временного исключения строки программы из обработки, то многострочные позволяют включать в программу длинный сопроводительный текст или пояснения.

Многострочный комментарий размещается между начальным (/*) и конечным (*/) ограничителями. PL/SQL рассматривает весь текст между этими двумя парами символов как часть комментария и игнорируется компилятором.

В следующем примере многострочный комментарий располагается в разделе заголовка процедуры. Столбик из вертикальных черт у левого края помогает зрительно выделить комментарий в программе:

```
PROCEDURE calc_revenue (company_id IN NUMBER) IS  
/*  
| Имя программы: calc_revenue  
| Автор: Стивен Фейерштейн  
| История изменений:  
|   10-06-2009 – введение новых формул  
|   23-10-2008 – создание программы  
|*/  
BEGIN  
...  
END;
```

С помощью многострочных комментариев также можно отменить выполнение части программного кода на время тестирования. В следующем примере дополнительные условия в операторе EXIT игнорируются для направленного тестирования функции a_delimiter:

```
EXIT WHEN a_delimiter (next_char)  
/*  
    OR  
    (was_a_delimiter AND NOT a_delimiter (next_char))  
*/  
;
```

Ключевое слово PRAGMA

Ключевое слово **PRAGMA** происходит из греческого языка, где оно означает «действие» или «операцию». В других языках программирования этим ключевым словом обычно помечаются строки исходного кода с описанием действий, которые должны быть приняты компилятором. По сути, ключевое слово **PRAGMA** является директивой, которая передает некоторую управляющую информацию компилятору, но не преобразуется непосредственно в исполняемый код.

Ключевое слово **PRAGMA** в PL/SQL имеет следующий синтаксис:

PRAGMA *директива*;

Компилятор PL/SQL позволяет размещать такие директивы в любой точке раздела объявлений, но для некоторых директив устанавливаются дополнительные требования к размещению. PL/SQL поддерживает следующие директивы:

- **AUTONOMOUS_TRANSACTION** — приказывает исполнительному ядру PL/SQL выполнить сохранение или откат любых изменений, внесенных в базу данных в текущем блоке, без воздействия на главную или внешнюю транзакцию. За дополнительной информацией обращайтесь к главе 14.
- **EXCEPTION_INIT** — приказывает компилятору связать конкретный номер ошибки с идентификатором, объявленным в программе как исключение. Идентификатор должен соответствовать правилам объявления исключений. За дополнительной информацией обращайтесь к главе 6.
- **RESTRICT_REFERENCES** — задает для компилятора уровень чистоты программного пакета (отсутствия действий, вызывающих побочные эффекты). За дополнительной информацией обращайтесь к главе 17.
- **SERIALLY_REUSABLE** — сообщает исполнительному ядру PL/SQL, что данные уровни пакета не должны сохраняться между обращениями к ним. За дополнительной информацией обращайтесь к главе 18.

Следующий блок демонстрирует применение директивы **EXCEPTION_INIT** для назначения имени встроенному исключению. Если этого не сделать, то исключение будет иметь только номер.

```
DECLARE
    no_such_sequence EXCEPTION;
    PRAGMA EXCEPTION_INIT (no_such_sequence, -2289);
BEGIN
    ...
EXCEPTION
    WHEN no_such_sequence
    THEN
        q$error_manager.raise_error ('Sequence not defined');
END;
```

Метки

Метки PL/SQL предназначены для присваивания имени определенной части программы. Синтаксис метки:

<<идентификатор>>

Здесь *идентификатор* — это допустимый идентификатор PL/SQL (длиной до 30 символов и начинающийся с буквы — см. раздел «Идентификатор»). Метка не имеет завершителя; она располагается непосредственно перед фрагментом кода, имя которого она определяет, даже если это простая команда **NULL**:

```
BEGIN
    ...
    <<the_spot>>
    NULL;
```

Поскольку анонимный блок представляет собой группу исполняемых команд, с помощью метки можно задать имя анонимного блока (на время его выполнения). Пример:

```
<<insert_but_ignore_dups>>
BEGIN
    INSERT INTO catalog
    VALUES (...);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX
    THEN
        NULL;
END insert_but_ignore_dups;
```

Блоки часто именуют для того, чтобы код лучше читался. Давая блоку имя, вы делаете код *самодокументируемым*. Заодно вы еще раз задумаетесь над тем, что он делает (иногда это даже помогает найти незамеченные ошибки). Метки используются также для уточнения ссылок на элементы внешнего блока, имена которых совпадают с именами текущего (вложенного) блока. Пример:

```
<<outerblock>>
DECLARE
    counter INTEGER := 0;
BEGIN
    ...
    DECLARE
        counter INTEGER := 1;
    BEGIN
        IF counter = outerblock.counter
        THEN
            ...
        END IF;
    END;
END;
```

Без метки блока невозможно различить две переменные с одинаковыми именами *counter* (впрочем, проблема лучше решается присваиванием разных имен).

У меток также есть третья функция: они могут служить целевыми точками перехода для команд *GOTO* (эта команда рассматривается в главе 4).

Хотя некоторые программы, с которыми мне довелось работать, использовали метки, последняя функция меток важнее всех трех предыдущих вместе взятых: метка может использоваться в качестве целевой точки оператора выхода из вложенных циклов *EXIT*:

```
BEGIN
    <<outer_loop>>
    LOOP
        LOOP
            EXIT outer_loop;
        END LOOP;
        команда;
    END LOOP;
END;
```

Без метки *<<outer_loop>>* команда *EXIT* осуществит выход только из внутреннего цикла, после чего будет выполнена *команда*, что совершенно излишне. Итак, в данном случае метка позволяла реализовать функциональность, которую в PL/SQL было бы затруднительно реализовать другим способом.



Структура программы PL/SQL

В этой части книги рассматриваются основы программирования и программные конструкции PL/SQL. В главах с 4 по 6 представлены условные команды (**IF** и **CASE**) и операторы управления последовательностью выполнения (**GOTO** и **NULL**), циклы и оператор **CONTINUE**, появившийся в Oracle11g, а также методы обработки исключений в PL/SQL. После знакомства с этой частью книги вы научитесь составлять блоки кода, отвечающие сложным требованиям ваших приложений.

4

Условные команды и переходы

В этой главе вы познакомитесь с двумя типами управляющих команд PL/SQL: условными командами и командами перехода. Команды первого типа, присутствующие почти во всех программах, управляют последовательностью выполнения программного кода в зависимости от заданных условий. В языке PL/SQL к этой категории относятся команды IF-THEN-ELSE и CASE. Также существуют так называемые CASE-выражения, которые иногда позволяют обойтись без команд IF и CASE. Значительно реже используются команды второго типа: GOTO (безусловный переход) и NULL (не выполняет никаких действий).

Команды IF

Команда IF реализует логику условного выполнения команд программы. С ее помощью можно реализовать конструкции следующего вида:

- Если оклад находится в пределах от \$10 000 до \$20 000, начислить премию в размере \$1500.
- Если коллекция содержит более 100 элементов, удалить лишнее.

Команда IF существует в трех формах, представленных в следующей таблице.

| Разновидность IF | Характеристики |
|----------------------------|--|
| IF THEN END IF; | Простейшая форма команды IF. Условие между IF и THEN определяет, должна ли выполняться группа команд, находящаяся между THEN и END IF. Если результат проверки условия равен FALSE или NULL, то код не выполняется |
| IF THEN ELSE END IF; | Реализация логики «или-или». В зависимости от условия между ключевыми словами IF и THEN выполняется либо код, находящийся между THEN и ELSE, либо код между ELSE и END IF. В любом случае выполняется только одна из двух групп исполняемых команд |
| IF THEN ELSIF ELSE END IF; | Последняя, и самая сложная, форма IF выбирает действие из набора взаимоисключающих условий и выполняет соответствующую группу исполняемых команд. Если вы пишете подобную конструкцию IF в версии Oracle9i Release 1 и выше, подумайте, не заменить ли ее командой выбора CASE |

Комбинация IF-THEN

Общий синтаксис конструкции IF-THEN выглядит так:

```
IF условие
THEN
... последовательность исполняемых команд ...
END IF;
```

Здесь *условие* — это логическая переменная, константа или логическое выражение с результатом TRUE, FALSE или NULL. Исполняемые команды между ключевыми словами THEN и END IF выполняются, если результат проверки условия равен TRUE, и не выполняются — если он равен FALSE или NULL.

ТРЕХЗНАЧНАЯ ЛОГИКА

Логические выражения могут возвращать три возможных результата. Когда все значения в логическом выражении известны, результат равен TRUE или FALSE. Например, истинность или ложность выражений вида

`(2 < 3) AND (5 < 10)`

сомнений не вызывает. Однако иногда оказывается, что некоторые значения в выражении неизвестны. Это может быть связано с тем, что соответствующие столбцы базы данных содержат NULL или остались пустыми. Каким должен быть результат выражений с NULL, например:

`2 < NULL`

Так как отсутствующее значение неизвестно, на этот вопрос можно дать только один ответ: «Неизвестно». В этом и заключается суть так называемой трехзначной логики — возможными результатами могут быть не только TRUE и FALSE, но и NULL.

Если вы захотите больше узнать о трехзначной логике, я рекомендую статью Лекса де Хаана и Джонатана Генника «Nulls: Nothing to Worry About» из Oracle Magazine. Также полезную информацию можно найти в книге С. Дж. Дейта «Database in Depth: Relational Theory for the Practitioner». Мы еще вернемся к трехзначной логике в этой главе.

Следующая условная команда IF сравнивает два числовых значения. Учтите, что если одно из них равно NULL, то и результат всего выражения равен NULL (если переменная salary равна NULL, то give_bonus не выполняется):

```
IF salary > 40000
THEN
    give_bonus (employee_id,500);
END IF;
```

У правила, согласно которому NULL в логическом выражении дает результат NULL, имеются исключения. Некоторые операторы и функции специально реализованы так, чтобы при работе с NULL они давали результаты TRUE и FALSE (но не NULL). Например, для проверки значения NULL можно воспользоваться конструкцией IS NULL:

```
IF salary > 40000 OR salary IS NULL
THEN
    give_bonus (employee_id,500);
END IF;
```

В этом примере условие salary IS NULL дает результат TRUE, если salary не содержит значения, и результат FALSE во всех остальных случаях.



Для обнаружения возможных значений NULL и их обработки удобно применять такие операторы, как IS NULL и IS NOT NULL, или функции COALESCE и NVL2. Для каждой переменной в каждом написанном вами логическом выражении подумайте, что произойдет, если эта переменная содержит NULL.

Ключевые слова IF, THEN и END IF не обязательно размещать в отдельных строках. В командах IF разрывы строк не важны, поэтому приведенный выше пример можно было бы записать так:

```
IF salary > 40000 THEN give_bonus (employee_id,500); END IF;
```

Размещение всей команды в одной строке отлично подходит для простых конструкций IF — таких, как в приведенном примере. Но любая хоть сколько-нибудь сложная команда гораздо лучше читается, когда каждое ключевое слово размещается в отдельной строке. Например, если записать следующий фрагмент в одну строку, в нем будет довольно трудно разобраться. В нем нелегко разобраться даже тогда, когда он записан в три строки:

```
IF salary > 40000 THEN INSERT INTO employee_bonus (eb_employee_id, eb_bonus_amt)
VALUES (employee_id, 500); UPDATE emp_employee SET emp_bonus_given=1 WHERE emp_
employee_id=employee_id; END IF;
```

И та же команда вполне нормально читается при разбиении на строки:

```
IF salary > 40000
THEN
    INSERT INTO employee_bonus
        (eb_employee_id, eb_bonus_amt)
    VALUES (employee_id, 500);
    UPDATE emp_employee
    SET emp_bonus_given=1
    WHERE emp_employee_id=employee_id;
END IF;
```

Вопрос удобочитаемости становится еще более важным при использовании ключевых слов ELSE и ELSIF, а также вложенных команд IF. Поэтому, чтобы сделать логику команд IF максимально наглядной, мы рекомендуем применять все возможности отступов и форматирования. И те программисты, которым придется сопровождать ваши программы, будут вам очень признательны.

Конструкция IF-THEN-ELSE

Конструкция IF-THEN-ELSE применяется при выборе одного из двух взаимоисключающих действий. Формат этой версии команды IF:

```
IF условие
THEN
    ... последовательность команд для результата TRUE ...
ELSE
    ... последовательность команд для результата FALSE/NULL ...
END IF;
```

Здесь *условие* — это логическая переменная, константа или логическое выражение. Если его значение равно TRUE, то выполняются команды, расположенные между ключевыми словами THEN и ELSE, а если FALSE или NULL — команды между ключевыми словами ELSE и END IF. Важно помнить, что в конструкции IF-THEN-ELSE *всегда* выполняется одна из двух возможных последовательностей команд. После выполнения соответствующей последовательности управление передается команде, которая расположена сразу после ключевых слов END IF.

Следующая конструкция IF-THEN-ELSE расширяет пример IF-THEN, приведенный в предыдущем разделе:

```
IF salary <= 40000
THEN
    give_bonus (employee_id, 0);
ELSE
    give_bonus (employee_id, 500);
END IF;
```


в этом примере сотрудники с окладом более 40 000 получают премию в 500, а остальным премия не назначается. Или все же назначается? Что произойдет, если у сотрудника по какой-либо причине оклад окажется равным NULL? В этом случае будут выполнены команды, следующие за ключевым словом ELSE, и работник получит премию, положенную только высокооплачиваемому составу. Поскольку мы не можем быть уверены в том, что оклад ни при каких условиях не окажется равным NULL, нужно защититься от подобных проблем при помощи функции NVL:

```
IF NVL(salary,0) <= 40000
THEN
  give_bonus (employee_id, 0);
ELSE
  give_bonus (employee_id, 500);
END IF;
```

Функция NVL возвращает нуль, если переменная salary равна NULL. Это гарантирует, что работникам с окладом NULL будет начислена нулевая премия (не позавидуешь!).

ЛОГИЧЕСКИЕ ФЛАГИ

Логические переменные удобно использовать в качестве флагов, чтобы одно и то же логическое выражение не приходилось вычислять по нескольку раз. Помните, что результат такого выражения можно присвоить логической переменной. Например, вместо

```
IF :customer.order_total > max_allowable_order
THEN
  order_exceeds_balance := TRUE;
ELSE
  order_exceeds_balance := FALSE;
END IF;
```

можно воспользоваться следующим, гораздо более простым выражением (при условии, что ни одна из переменных не равна NULL):

```
order_exceeds_balance
:= :customer.order_total > max_allowable_order;
```

Теперь если где-либо в программном коде потребуется проверить, не превышает ли сумма заказа (order_total) максимально допустимое значение (max_allowable_order), достаточно простой и понятной конструкции IF:

```
IF order_exceeds_balance
THEN
  ...
```

Если вам еще не приходилось работать с логическими переменными, возможно, на освоение этих приемов уйдет некоторое время. Но затраты окупятся сполна, поскольку в результате вы получите более простой и понятный код.

Конструкция IF-THEN-ELSIF

Данная форма команды IF удобна для реализации логики с несколькими альтернативными действиями в одной команде IF. Как правило, ELSIF используется с взаимоисключающими альтернативами (то есть при выполнении команды IF истинным может быть только одно из условий). Обобщенный синтаксис этой формы IF выглядит так:

```
IF условие-1
THEN
  команды-1
ELSIF condition-N
```

```

THEN
    команды-N
[ELSE
    команды_else]
END IF;

```



Некоторые программисты пытаются записывать **ELSIF** в виде **ELSEIF** или **ELSE IF**. Это очень распространенная синтаксическая ошибка.

Формально конструкция **IF-THEN-ELSIF** представляет собой один из способов реализации функций команды **CASE** в **PL/SQL**. Конечно, если вы используете **Oracle9i**, лучше воспользоваться командой **CASE**, о которой будет рассказано далее в этой главе.

В каждой секции **ELSIF** (кроме секции **ELSE**) *за условием* должно следовать ключевое слово **THEN**. Секция **ELSE** в **IF-ELSIF** означает «если не выполняется ни одно из условий», то есть когда ни одно из условий не равно **TRUE**, выполняются команды, следующие за **ELSE**. Следует помнить, что секция **ELSE** не является обязательной — конструкция **IF-ELSIF** может состоять только из секций **IF** и **ELSIF**. Если ни одно из условий не равно **TRUE**, то никакие команды блока **IF** не выполняются.

Далее приводится полная реализация логики назначения премий, описанной в начале главы, на базе конструкции **IF-THEN-ELSEIF**:

```

IF salary BETWEEN 10000 AND 20000
THEN
    give_bonus(employee_id, 1500);
ELSIF salary BETWEEN 20000 AND 40000
THEN
    give_bonus(employee_id, 1000);
ELSIF salary > 40000
THEN
    give_bonus(employee_id, 500);
ELSE
    give_bonus(employee_id, 0);
END IF;

```

Ловушки синтаксиса IF

Запомните несколько правил, касающихся применения команды **IF**:

- **Каждая команда IF должна иметь парную конструкцию END IF.** Все три разновидности данной команды обязательно должны явно закрываться ключевым словом **END IF**.
- **Не забывайте разделять пробелами ключевые слова END и IF.** Если вместо **END IF** ввести **ENDIF**, компилятор выдаст малопонятное сообщение об ошибке:


```
ORA-06550: line 14, column 4:
PLS-00103: Encountered the symbol ";" when expecting one of the following:
```
- **Ключевое слово ELSIF должно содержать только одну букву «Е».** Если вместо ключевого слова **ELSIF** указать **ELSEIF**, компилятор не воспримет последнее как часть команды **IF**. Он интерпретирует его как имя переменной или процедуры.
- **Точка с запятой ставится только после ключевых слов END IF.** После ключевых слов **THEN**, **ELSE** и **ELSIF** точка с запятой не ставится. Они не являются отдельными исполняемыми командами и, в отличие от **END IF**, не могут завершать команду **PL/SQL**. Если вы все же поставите точку с запятой после этих ключевых слов, компилятор выдаст сообщение об ошибке.

Условия IF-ELSIF всегда обрабатываются от первого к последнему. Если оба условия равны TRUE, то выполняются команды первого условия. В контексте текущего примера для оклада \$20 000 будет начислена премия \$1500, хотя оклад \$20 000 также удовлетворяет условию премии \$1000 (проверка BETWEEN включает границы). Если какое-либо условие истинно, остальные условия вообще не проверяются.

Команда CASE позволяет решить задачу начисления премии более элегантно, чем решение IF-THEN-ELSIF в этом разделе (см. раздел «Команды и выражения CASE»).

И хотя в команде IF-THEN-ELSIF разрешены перекрывающиеся условия, лучше избегать их там, где это возможно. В моем примере исходная спецификация немного неоднозначна в отношении граничных значений (таких, как 20 000). Если предположить, что работникам с низшими окладами должны начисляться более высокие премии (что на мой взгляд вполне разумно), я бы избавился от а BETWEEN и воспользовался логикой «меньше/больше» (см. далее). Также обратите внимание на отсутствие секции ELSE — я опустил ее просто для того, чтобы показать, что она не является обязательной:

```
IF salary >= 10000 AND salary <= 20000
THEN
    give_bonus(employee_id, 1500);
ELSIF salary > 20000 AND salary <= 40000
THEN
    give_bonus(employee_id, 1000);
ELSIF salary > 40000
THEN
    give_bonus(employee_id, 400);
END IF;
```

Принимая меры к предотвращению перекрывающихся условий в IF-THEN-ELSIF, я устраняю возможный (и даже вероятный) источник ошибок для программистов, которые будут работать с кодом после меня. Я также устраняю возможность введения случайных ошибок в результате переупорядочения секций ELSIF. Однако следует заметить, что если значение salary равно NULL, никакой код выполнен не будет, потому что секции ELSE отсутствуют.

Язык не требует, чтобы условия ELSIF были взаимоисключающими. Всегда учитывайте вероятность того, что значение может подходить по двум и более условиям, поэтому порядок условий ELSIF может быть важен.

Вложенные команды IF

Команды IF можно вкладывать друг в друга. В следующем примере представлены команды IF с несколькими уровнями вложенности:

```
IF условие1
THEN
    IF условие2
    THEN
        команды2
    ELSE
        IF условие3
        THEN
            команды3
        ELSIF условие4
        THEN
            команды4
        END IF;
    END IF;
END IF;
```

Сложную логику часто невозможно реализовать без вложенных команд **IF**, но их использование требует крайней осторожности. Вложенные команды **IF**, как и вложенные циклы, затрудняют чтение программы и ее отладку. И если вы собираетесь применить команды **IF** более чем с тремя уровнями вложения, подумайте, нельзя ли пересмотреть логику программы и реализовать требования более простым способом. Если такового не найдется, подумайте о создании одного или нескольких локальных модулей, скрывающих внутренние команды **IF**. Главное преимущество вложенных структур **IF** заключается в том, что они позволяют отложить проверку внутренних условий. Условие внутренней команды **IF** проверяется только в том случае, если значение выражения во внешнем условии равно **TRUE**. Таким образом, очевидной причиной вложения команд **IF** может быть проверка внутреннего условия только при истинности другого. Например, код начисления премий можно было бы записать так:

```
IF award_bonus(employee_id) THEN
  IF print_check (employee_id) THEN
    DBMS_OUTPUT.PUT_LINE('Check issued for ' || employee_id);
  END IF;
END IF;
```

Такая реализация вполне разумна, потому что для каждой начисленной премии должно выводиться сообщение, но если премия не начислялась, сообщение с нулевой суммой выводиться не должно.

Ускоренное вычисление

В PL/SQL используется *ускоренное вычисление условий*; иначе говоря, вычислять все выражения в условиях **IF** не обязательно. Например, при вычислении выражения в следующей конструкции **IF** PL/SQL прекращает обработку и немедленно выполняет ветвь **ELSE**, если первое условие равно **FALSE** или **NULL**:

```
IF условие1 AND условие2
THEN
  ...
ELSE
  ...
END IF;
```

PL/SQL прерывает вычисление, если *условие_1* равно **FALSE** или **NULL**, потому что ветвь **THEN** выполняется только в случае истинности всего выражения, а для этого оба подвыражения должны быть равны **TRUE**. Как только обнаруживается, что хотя бы одно подвыражение отлично от **TRUE**, дальнейшие проверки излишни — ветвь **THEN** все равно выбрана не будет.



Изучая поведение ускоренного вычисления в PL/SQL, я обнаружил нечто интересное: его поведение зависит от контекста выражения. Возьмем следующую команду:

```
my_boolean := condition1 AND condition2
```

В отличие от случаев с командой **IF**, если *условие1* равно **NULL**, ускоренное вычисление применяться не будет. Почему? Потому что результат может быть равен **NULL** или **FALSE** в зависимости от условия2. Для команды **IF** оба значения **NULL** и **FALSE** ведут к ветви **ELSE**, поэтому ускоренное вычисление возможно. Но для присваивания должно быть известно конечное значение, и ускоренное вычисление в этом случае может (и будет) происходить только в том случае, если *условие1* равно **FALSE**.

Аналогичным образом работает ускоренное вычисление в операциях **OR**: если первый операнд **OR** в конструкции **IF** равен **TRUE**, PL/SQL немедленно выполняет ветвь **THEN**:

```
IF условие1 OR условие2
THEN
...
ELSE
...
END IF;
```

Ускоренное вычисление может быть полезно в том случае, если одно из условий требует особенно серьезных затрат ресурсов процессора или памяти. Такие условия следует размещать в конце составного выражения:

```
IF простое_условие AND сложное_условие
THEN
...
END IF;
```

Сначала проверяется *простое_условие*, и если его результата достаточно для определения конечного результата операции AND (то есть если результат равен FALSE), более затратное условие не проверяется, а пропущенная проверка улучшает быстродействие приложения.



Но если работа вашей программы зависит от вычисления второго условия — например, из-за побочных эффектов от вызова хранимой функции, вызываемой в условии, — значит, вам придется пересмотреть структуру кода. Я считаю, что такая зависимость от побочных эффектов нежелательна.

Ускоренное вычисление легко имитируется при помощи вложения IF:

```
IF низкозатратное_условие
THEN
  IF высокозатратное_условие
  THEN
    ...
  END IF;
END IF;
```

Сложное_условие проверяется только в том случае, если *простое_условие* окажется истинным. Происходит то же, что при ускоренном вычислении, но зато при чтении программы можно с первого взгляда сказать, что же в ней происходит. Кроме того, сразу понятно, что в соответствии с намерениями программиста *простое_условие* должно проверяться первым.

Ускоренное вычисление также применяется к командам и выражениям CASE, описанным в следующем разделе.

Команды и выражения CASE

Команда CASE позволяет выбрать для выполнения одну из нескольких последовательностей команд. Эта конструкция присутствует в стандарте SQL с 1992 года, хотя в Oracle SQL она не поддерживалась вплоть до версии Oracle8i, а в PL/SQL — до версии Oracle9i Release 1. Начиная с этой версии, поддерживаются следующие разновидности команд CASE:

- Простая команда CASE — связывает одну или несколько последовательностей команд PL/SQL с соответствующими значениями (выполняемая последовательность выбирается с учетом результата вычисления выражения, возвращающего одно из значений).
- Поисковая команда CASE — выбирает для выполнения одну или несколько последовательностей команд в зависимости от результатов проверки списка логических значений. Выполняется последовательность команд, связанная с первым условием, результат проверки которого оказался равным TRUE.

NULL ИЛИ UNKNOWN?

Ранее я указал, что результат логического выражения может быть равен TRUE, FALSE или NULL.

В PL/SQL это утверждение истинно, но в более широком контексте реляционной теории считается некорректным говорить о возврате NULL из логического выражения. Реляционная теория говорит, что сравнение с NULL следующего вида:

```
2 < NULL
```

дает логический результат UNKNOWN, причем значение UNKNOWN не эквивалентно NULL. Впрочем, вам не стоит особенно переживать из-за того, что в PL/SQL для UNKNOWN используется обозначение NULL. Однако вам следует знать, что третьим значением в трехзначной логике является UNKNOWN. И я надеюсь, что вы никогда не попадете впросак (как это бывало со мной!), используя неправильный термин при обсуждении трехзначной логики с экспертами в области реляционной теории.

Кроме команд CASE, PL/SQL также поддерживает *CASE-выражения*. Такое выражение очень похоже на команду CASE, оно позволяет выбрать для вычисления одно или несколько выражений. Результатом выражения CASE является одно значение, тогда как результатом команды CASE является выполнение последовательности команд PL/SQL.

Простые команды CASE

Простая команда CASE позволяет выбрать для выполнения одну из нескольких последовательностей команд PL/SQL в зависимости от результата вычисления выражения. Он записывается следующим образом:

```
CASE выражение
WHEN результат_1 THEN
    команды_1
WHEN результат_2 THEN
    команды_2
...
ELSE
    команды_else
END CASE;
```

Ветвь ELSE здесь не обязательна. При выполнении такой команды PL/SQL сначала вычисляет *выражение*, после чего результат сравнивается с *результат_1*. Если они совпадают, то выполняются *команды_1*. В противном случае проверяется значение *результат_2* и т. д.

Приведем пример простой команды CASE, в котором премия начисляется в зависимости от значения переменной *employee_type*:

```
CASE employee_type
WHEN 'S' THEN
    award_salary_bonus(employee_id);
WHEN 'H' THEN
    award_hourly_bonus(employee_id);
WHEN 'C' THEN
    award_commissioned_bonus(employee_id);
ELSE
    RAISE invalid_employee_type;
END CASE;
```

В этом примере присутствует явно заданная секция `ELSE`, однако в общем случае она не обязательна. Без секции `ELSE` компилятор PL/SQL неявно подставляет такой код:

```
ELSE
  RAISE CASE_NOT_FOUND;
```

Иначе говоря, если не задать ключевое слово `ELSE` и если никакой из результатов в секциях `WHEN` не соответствует результату выражения в команде `CASE`, PL/SQL инициирует исключение `CASE_NOT_FOUND`. В этом и заключается отличие данной команды от `IF`. Когда в команде `IF` отсутствует ключевое слово `ELSE`, то при невыполнении условия не происходит ничего, тогда как в команде `CASE` аналогичная ситуация приводит к ошибке.

Интересно посмотреть, как с помощью простой команды `CASE` реализовать описанную в начале главы логику начисления премий. На первый взгляд это кажется невозможным, но подойдя к делу творчески, мы приходим к следующему решению:

```
CASE TRUE
WHEN salary >= 10000 AND salary <= 20000
THEN
  give_bonus(employee_id, 1500);
WHEN salary > 20000 AND salary <= 40000
THEN
  give_bonus(employee_id, 1000);
WHEN salary > 40000
THEN
  give_bonus(employee_id, 500);
ELSE
  give_bonus(employee_id, 0);
END CASE;
```

Здесь важно то, что элементы *выражение* и *результат* могут быть либо скалярными значениями, либо выражениями, результатами которых являются скалярные значения.

Вернувшись к команде `IF...THEN...ELSIF`, реализующей ту же логику, вы увидите, что в команде `CASE` определена секция `ELSE`, тогда как в команде `IF-THEN-ELSIF` ключевое слово `ELSE` отсутствует. Причина добавления `ELSE` проста: если ни одно из условий начисления премии не выполняется, команда `IF` ничего не делает, и премия получается нулевой. Команда `CASE` в этом случае выдает ошибку, поэтому ситуацию с нулевым размером премии приходится программировать явно.



Чтобы предотвратить ошибки `CASE_NOT_FOUND`, убедитесь в том, что при любом значении проверяемого выражения будет выполнено хотя бы одно из условий.

Приведенная выше команда `CASE TRUE` кому-то покажется эффективным трюком, но на самом деле она всего лишь реализует поисковую команду `CASE`, о которой мы поговорим в следующем разделе.

Поисковая команда CASE

Поисковая команда `CASE` проверяет список логических выражений; обнаружив выражение, равное `TRUE`, выполняет последовательность связанных с ним команд. В сущности, поисковая команда `CASE` является аналогом команды `CASE TRUE`, пример которой приведен в предыдущем разделе. Поисковая команда `CASE` имеет следующую форму записи:

```
CASE
WHEN выражение_1 THEN
  команды_1
```

продолжение ➤

```
WHEN выражение_2 THEN
    команда_2
...
ELSE
    команды_else
END CASE;
```

Она идеально подходит для реализации логики начисления премии:

```
CASE
WHEN salary >= 10000 AND salary <=20000 THEN
    give_bonus(employee_id, 1500);
WHEN salary > 20000 AND salary <= 40000 THEN
    give_bonus(employee_id, 1000);
WHEN salary > 40000 THEN
    give_bonus(employee_id, 500);
ELSE
    give_bonus(employee_id, 0);
END CASE;
```

Поисковая команда CASE, как и простая команда, подчиняется следующим правилам:

- Выполнение команды заканчивается сразу же после выполнения последовательности исполняемых команд, связанных с истинным выражением. Если истинными оказываются несколько выражений, то выполняются команды, связанные с первым из них.
- Ключевое слово ELSE не обязательно. Если оно не задано и ни одно из выражений не равно TRUE, инициируется исключение CASE_NOT_FOUND.
- Условия WHEN проверяются в строго определенном порядке, от начала к концу.

Рассмотрим еще одну реализацию логики начисления премии, в которой используется то обстоятельство, что условия WHEN проверяются в порядке их записи. Отдельные выражения проще, но можно ли сказать, что смысл всей команды стал более понятным?

```
CASE
WHEN salary > 40000 THEN
    give_bonus(employee_id, 500);
WHEN salary > 20000 THEN
    give_bonus(employee_id, 1000);
WHEN salary >= 10000 THEN
    give_bonus(employee_id, 1500);
ELSE
    give_bonus(employee_id, 0);
END CASE;
```

Если оклад некоего сотрудника равен 20 000, то первые два условия равны FALSE, а третье — TRUE, поэтому сотрудник получит премию в 1500 долларов. Если же оклад равен 21 000, то результат второго условия будет равен TRUE, и премия составит 1000 долларов. Выполнение команды CASE завершится на второй ветви WHEN, а третье условие даже не будет проверяться. Стоит ли использовать такой подход при написании команд CASE — вопрос спорный. Как бы то ни было, имейте в виду, что написать такую команду возможно, а при отладке и редактировании программ, в которых результат зависит от порядка следования выражений, необходима особая внимательность.

Логика, зависящая от порядка следования однородных ветвей WHEN, является потенциальным источником ошибок, возникающих при их перестановке. В качестве примера рассмотрим следующую поисковую команду CASE, в которой при значении salary, равном 20 000, проверка условий в обеих ветвях WHEN дает TRUE:

```
CASE
WHEN salary BETWEEN 10000 AND 20000 THEN
    give_bonus(employee_id, 1500);
WHEN salary BETWEEN 20000 AND 40000 THEN
```



```
give_bonus(employee_id, 1000);
...
```

Представьте, что программист, занимающийся сопровождением этой программы, легкомысленно переставит ветви **WHEN**, чтобы упорядочить их по убыванию **salary**. Не отвергайте такую возможность! Программисты часто склонны «доводить до ума» прекрасно работающий код, руководствуясь какими-то внутренними представлениями о порядке. Команда **CASE** с переставленными секциями **WHEN** выглядит так:

```
CASE
WHEN salary BETWEEN 20000 AND 40000 THEN
    give_bonus(employee_id, 1000);
WHEN salary BETWEEN 10000 AND 20000 THEN
    give_bonus(employee_id, 1500);
...
```

На первый взгляд все верно, не так ли? К сожалению, из-за перекрытия двух ветвей **WHEN** в программе появляется коварная ошибка. Теперь сотрудник с окладом 20 000 получит премию 1000 вместо положенных 1500. Возможно, в некоторых ситуациях перекрытие между ветвями **WHEN** желательно и все же его следует по возможности избегать. Всегда помните, что порядок следования ветвей важен, и сдерживайте желание доработать уже работающий код — *«не чините то, что не сломано»*.



Поскольку условия **WHEN** проверяются по порядку, можно немного повысить эффективность кода, поместив ветви с наиболее вероятными условиями в начало списка. Кроме того, если у вас есть ветвь с «затратными» выражениями (например, требующими значительного процессорного времени и памяти), их можно поместить в конец, чтобы свести к минимуму вероятность их проверки. За подробностями обращайтесь к разделу «Вложенные команды **IF**».

Поисковые команды **CASE** используются в тех случаях, когда выполняемые команды определяются набором логических выражений. Простая команда **CASE** используется тогда, когда решение принимается на основании результата одного выражения.

Вложенные команды CASE

Команды **CASE**, как и команды **IF**, могут быть вложенными. Например, вложенная команда **CASE** присутствует в следующей (довольно запутанной) реализации логики начисления премий:

```
CASE
WHEN salary >= 10000 THEN
    CASE
        WHEN salary <= 20000 THEN
            give_bonus(employee_id, 1500);
        WHEN salary > 40000 THEN
            give_bonus(employee_id, 500);
        WHEN salary > 20000 THEN
            give_bonus(employee_id, 1000);
        END CASE;
WHEN salary < 10000 THEN
    give_bonus(employee_id, 0);
END CASE;
```

В команде **CASE** могут использоваться любые команды, так что внутренняя команда **CASE** легко заменяется командой **IF**. Аналогичным образом, в команду **IF** может быть вложена любая команда, в том числе и **CASE**.

Выражения CASE

Выражения CASE решают ту же задачу, что и команды CASE, но только не для исполняемых команд, а для выражений. Простое выражение CASE выбирает для вычисления одно из нескольких выражений на основании заданного скалярного значения. Поисковое выражение CASE последовательно вычисляет выражения из списка, пока одно из них не окажется равным TRUE, а затем возвращает результат связанного с ним выражения.

Синтаксис этих двух разновидностей выражений CASE:

```
Простое_выражение_Case :=
CASE выражение
WHEN результат_1 THEN
    результирующее_выражение_1
WHEN результат_2 THEN
    результирующее_выражение_2
...
ELSE
    результирующее_выражение_else
END;
Поисковое_выражение_Case :=
CASE
WHEN выражение_1 THEN
    результирующее_выражение_1
WHEN выражение_2 THEN
    результирующее_выражение_2
...
ELSE
    результирующее_выражение_else
END;
```

Выражение CASE возвращает одно значение — результат выбранного для вычисления выражения. Каждой ветви WHEN должно быть поставлено в соответствие одно результирующее выражение (но не команда). В конце выражения CASE не ставится ни точка с запятой, ни END CASE. Выражение CASE завершается ключевым словом END.

Далее приводится пример простого выражения CASE, используемого совместно с процедурой PUT_LINE пакета DBMS_OUTPUT для вывода на экран значения логической переменной. (Напомним, что программа PUT_LINE не поддерживает логические типы напрямую.) В этом примере выражение CASE преобразует логическое значение в символьную строку, которая затем выводится процедурой PUT_LINE:

```
DECLARE
    boolean_true BOOLEAN := TRUE;
    boolean_false BOOLEAN := FALSE;
    boolean_null BOOLEAN;
    FUNCTION boolean_to_varchar2 (flag IN BOOLEAN) RETURN VARCHAR2 IS
    BEGIN
        RETURN
            CASE flag
                WHEN TRUE THEN 'True'
                WHEN FALSE THEN 'False'
                ELSE 'NULL'
            END;
    END;
END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_true));
    DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_false));
    DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_null));
END;
```

Для реализации логики начисления премий можно использовать поисковое выражение CASE, возвращающее величину премии для заданного оклада:

```

DECLARE
  salary NUMBER := 20000;
  employee_id NUMBER := 36325;
  PROCEDURE give_bonus (emp_id IN NUMBER, bonus_amt IN NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emp_id);
    DBMS_OUTPUT.PUT_LINE(bonus_amt);
  END;
BEGIN
  give_bonus(employee_id,
    CASE
      WHEN salary >= 10000 AND salary <= 20000 THEN 1500
      WHEN salary > 20000 AND salary <= 40000 THEN 1000
      WHEN salary > 40000 THEN 500
      ELSE 0
    END);
END;

```

Выражение CASE может применяться везде, где допускается использование выражений любого другого типа. В следующем примере CASE-выражение используется для вычисления размера премии, умножения его на 10 и присваивания результата переменной, выводимой на экран средствами DBMS_OUTPUT:

```

DECLARE
  salary NUMBER := 20000;
  employee_id NUMBER := 36325;
  bonus_amount NUMBER;
BEGIN
  bonus_amount :=
    CASE
      WHEN salary >= 10000 AND salary <= 20000 THEN 1500
      WHEN salary > 20000 AND salary <= 40000 THEN 1000
      WHEN salary > 40000 THEN 500
      ELSE 0
    END * 10;
  DBMS_OUTPUT.PUT_LINE(bonus_amount);
END;

```

В отличие от команды CASE, если условие ни одной ветви WHEN не выполнено, выражение CASE не выдает ошибку, а просто возвращает NULL.

Команда GOTO

Команда GOTO выполняет безусловный переход к другой исполняемой команде в том же исполняемом разделе блока PL/SQL. Как и в случае с другими конструкциями языка, уместное и осторожное применение GOTO способно расширить возможности ваших программ.

Общий формат команды GOTO:

```
GOTO имя_метки;
```

Здесь *имя_метки* — имя метки, идентифицирующей целевую точку перехода.

В программе метка выглядит так:

```
<<имя_метки>>
```

Имя метки заключается в двойные угловые скобки (<< >>). Когда компилятор PL/SQL встречает команду GOTO, он немедленно передает управление первой исполняемой команде, следующей за меткой. Блок кода в следующем примере содержит и метку, и команду GOTO:

```

BEGIN
  GOTO second_output;

  DBMS_OUTPUT.PUT_LINE('Эта строка никогда не выполняется.');
```

```
<<second_output>>
DBMS_OUTPUT.PUT_LINE('Мы здесь!');
END;
```

На использование команды **GOTO** налагаются некоторые ограничения:

- За меткой должна следовать хотя бы одна исполняемая команда.
- Целевая метка должна находиться в пределах области действия оператора **GOTO**.
- Целевая метка должна находиться в той же части блока PL/SQL, что и оператор **GOTO**.

Несмотря на стойкое предубеждение против команды **GOTO**, иногда она бывает очень полезна. Например, в некоторых случаях она способна упростить логику программы. С другой стороны, поскольку PL/SQL включает так много разнообразных управляющих конструкций и средств модульного разбиения программного кода, практически для любой задачи можно найти более удобное решение, не связанное с использованием **GOTO**.

Команда NULL

Каждая команда программы, как правило, выполняет некоторое действие. Однако бывают случаи, когда нужно просто указать компилятору PL/SQL, чтобы он не делал ничего, и тогда на помощь приходит оператор **NULL**. Он имеет следующий формат:

```
NULL;
```

С чего бы команда, которая «ничего не делает», имела сложную структуру? Команда **NULL** состоит из ключевого слова **NULL**, за которым следует точка с запятой (;) — она указывает, что это команда, а не значение **NULL**. В программе эта команда не делает ничего, если не считать передачи управления следующей исполняемой команде.

О том, для чего может понадобиться такая команда, вы узнаете из следующих разделов.

Удобочитаемость кода

Иногда бывает полезно полностью исключить всякую неоднозначность, которая присутствует в командах **IF**, не покрывающих всех возможных случаев. Например, в команде **IF** может отсутствовать ключевое слово **ELSE** — как в следующем примере:

```
IF :report_mgr.selection = 'DETAIL'
THEN
    exec_detail_report;
END IF;
```

Что должна делать программа при выборе другого отчета вместо 'DETAIL'? Можно предположить, что ничего. Но поскольку в коде это явно не указано, кто-то может подумать, что вы просто забыли запрограммировать соответствующее действие. С другой стороны, присутствие секции **ELSE**, которая не делает ничего, предельно четко указывает: «Эта возможность предусмотрена, здесь действительно не должны выполняться никакие операции»:

```
IF :report_mgr.selection = 'DETAIL'
THEN
    exec_detail_report;
ELSE
    NULL; -- Ничего не делать
END IF;
```

Приведенный пример демонстрирует команду **IF**, но аналогичный принцип действует и при записи команд и выражений **CASE**. Если вам потребуется временно исключить весь код из функции или процедуры, но при этом оставить вызов функции или процедуры

в программе, используйте команду NULL в качестве временного «заполнителя». Без нее вы не сможете откомпилировать функцию или процедуру, не содержащую ни одной строки кода.

Использование NULL после метки

В некоторых случаях переход к NULL позволяет избежать выполнения дополнительных команд. Большинство программистов никогда не использует GOTO, а ситуации, в которых эта команда действительно необходима, крайне редки. Но если вам когда-нибудь придется использовать команду GOTO, помните, что за меткой, к которой осуществляется переход, должна стоять хотя бы одна исполняемая команда. В следующем примере команда GOTO используется для быстрого перехода в конец программы в том случае, если состояние данных указывает, что дальнейшая обработка не требуется:

```
PROCEDURE process_data (data_in IN orders%ROWTYPE,
                        data_action IN VARCHAR2)
IS
    status INTEGER;
BEGIN
    -- Первая проверка
    IF data_in.ship_date IS NOT NULL
    THEN
        status := validate_shipdate (data_in.ship_date);
        IF status != 0 THEN GOTO end_of_procedure; END IF;
    END IF;

    -- Вторая проверка
    IF data_in.order_date IS NOT NULL
    THEN
        status := validate_orderdate (data_in.order_date);
        IF status != 0 THEN GOTO end_of_procedure; END IF;
    END IF;

    ... Дополнительные проверки ...

    <<end_of_procedure>>
    NULL;
END;
```

При обнаружении ошибки в одном из разделов остальные проверки обходятся командой GOTO. Поскольку в конце процедуры делать ничего не нужно, но там должна находиться хотя бы одна исполняемая команда, после метки помещается NULL. Хотя последняя никаких реальных действий не выполняет, она считается исполняемым оператором.

5 Циклы

В этой главе рассматриваются управляющие структуры PL/SQL, называемые *циклами* и предназначенные для многократного выполнения программного кода. Также мы рассмотрим команду `CONTINUE`, появившуюся в Oracle11g. PL/SQL поддерживает циклы трех видов: простые (бесконечные), `FOR` и `WHILE`. Каждая разновидность циклов предназначена для определенных целей, имеет свои нюансы и правила использования. Из представленных ниже таблиц вы узнаете, как завершается цикл, когда проверяется условие его завершения и в каких случаях применяются циклы того или иного вида.

| Свойство | Описание |
|--|--|
| Условие завершения цикла | Код выполняется многократно. Как остановить выполнение тела цикла? |
| Когда проверяется условие завершения цикла | Когда выполняется проверка условия завершения — в начале или в конце цикла? К каким последствиям это приводит? |
| В каких случаях используется данный цикл | Какие специальные факторы необходимо учитывать, если цикл подходит для вашей ситуации? |

Основы циклов

Зачем нужны три разновидности циклов? Чтобы вы могли выбрать оптимальный способ решения каждой конкретной задачи. В большинстве случаев задачу можно решить с помощью любой из трех циклических конструкций, но при неудачном выборе конструкции вам придется написать множество лишних строк программного кода, а это затруднит понимание и сопровождение написанных модулей.

Примеры разных циклов

Чтобы дать начальное представление о разных циклах и о том, как они работают, рассмотрим три процедуры. В каждом случае для каждого года в диапазоне от начального до конечного значения вызывается процедура `display_total_sales`.

Простой цикл начинается с ключевого слова `LOOP` и завершается командой `END LOOP`. Выполнение цикла прерывается при выполнении команды `EXIT`, `EXIT WHEN` или `RETURN` в теле цикла (или при возникновении исключения):

```

/* Файл в Сети: loop_examples.sql
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
    ,end_year_in IN PLS_INTEGER
)
IS
    l_current_year PLS_INTEGER := start_year_in;
BEGIN
    LOOP
        EXIT WHEN l_current_year > end_year_in;
        display_total_sales (l_current_year);
        l_current_year := l_current_year + 1;
    END LOOP;
END display_multiple_years;

```

Цикл FOR существует в двух формах: числовой и курсорной. В числовых циклах FOR программист задает начальное и конечное целочисленные значения, а PL/SQL перебирает все промежуточные значения, после чего завершает цикл:

```

/* Файл в Сети: loop_examples.sql
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
    ,end_year_in IN PLS_INTEGER
)
IS
BEGIN
    FOR l_current_year IN start_year_in .. end_year_in
    LOOP
        display_total_sales (l_current_year);
    END LOOP;
END display_multiple_years;

```

Курсорная форма цикла FOR имеет аналогичную базовую структуру, но вместо границ числового диапазона в ней задается курсор или конструкция SELECT:

```

/* Файл в Сети: loop_examples.sql
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
    ,end_year_in IN PLS_INTEGER
)
IS
BEGIN
    FOR l_current_year IN (
        SELECT * FROM sales_data
        WHERE year BETWEEN start_year_in AND end_year_in)
    LOOP
        -- Процедура передается запись, неявно объявленная
        -- с типом sales_data%ROWTYPE...
        display_total_sales (l_current_year);
    END LOOP;
END display_multiple_years;

```

Цикл WHILE имеет много общего с простым циклом. Принципиальное отличие заключается в том, что условие завершения проверяется перед выполнением очередной итерации. Возможны ситуации, в которых тело цикла не будет выполнено ни одного раза:

```

/* Файл в Сети: loop_examples.sql
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
    ,end_year_in IN PLS_INTEGER
)
IS
    l_current_year PLS_INTEGER := start_year_in;
BEGIN
    WHILE (l_current_year <= end_year_in)
    LOOP
        display_total_sales (l_current_year);
    END LOOP;
END display_multiple_years;

```

продолжение ➤

```

    l_current_year := l_current_year + 1;
END LOOP;
END display_multiple_years;

```

В приведенных примерах самым компактным получился цикл **FOR**. Однако его можно использовать только потому, что нам заранее известно, сколько раз будет выполняться тело цикла. Во многих других случаях количество повторений может быть заранее неизвестно, поэтому для них цикл **FOR** не подходит.

Структура циклов PL/SQL

Несмотря на различия между разными формами циклических конструкций, каждый цикл состоит из двух частей: ограничителей и тела цикла.

Ограничители — ключевые слова, определяющие начало цикла, условие завершения, и команда **END LOOP**, завершающая цикл. *Тело цикла* — последовательность исполняемых команд внутри границ цикла, выполняемых на каждой итерации.

На рис. 5.1 изображена структура цикла **WHILE**.

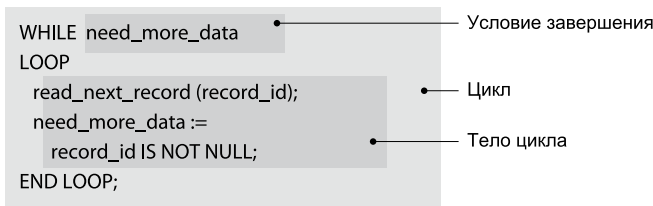


Рис. 5.1. Цикл **WHILE** и его тело

В общем случае цикл можно рассматривать как процедуру или функцию. Его тело — своего рода «черный ящик», а условие завершения — это интерфейс «черного ящика». Код, находящийся вне цикла, ничего не должен знать о происходящем внутри него. Помните об этом при рассмотрении различных форм циклов в этой главе.

Простой цикл

Структура простого цикла является самой элементарной среди всех циклических конструкций. Такой цикл состоит из ключевого слова **LOOP**, исполняемого кода (тела цикла) и ключевых слов **END LOOP**:

```

LOOP
  исполняемые_команды
END LOOP;

```

Цикл начинается командой **LOOP**, а заканчивается командой **END LOOP**. Тело цикла должно содержать как минимум одну исполняемую команду. Свойства простого цикла описаны в следующей таблице.

| Свойство | Описание |
|--|---|
| Условие завершения цикла | Если в теле цикла выполняется команда EXIT . В противном случае цикл выполняется бесконечно |
| Когда проверяется условие завершения цикла | В теле цикла и только при выполнении команды EXIT или EXIT WHEN . Таким образом, тело цикла (или его часть) всегда выполняется как минимум один раз |
| В каких случаях используется данный цикл | (1) Если не известно, сколько раз будет выполняться тело цикла; (2) тело цикла должно быть выполнено хотя бы один раз |

Простой цикл удобно использовать, когда нужно гарантировать хотя бы однократное выполнение тела цикла (или хотя бы его части). Так как цикл не имеет условия, которое бы определяло, должен ли он выполняться или нет, тело цикла всегда будет выполнено хотя бы один раз.

Простой цикл завершается только в том случае, если в его теле выполняется команда `EXIT` (или ее «близкий родственник» — `EXIT WHEN`) или же в нем инициируется исключение (оставшееся необработанным).

Завершение простого цикла: `EXIT` и `EXIT WHEN`

Если вы не хотите, чтобы программа «зациклилась», в теле цикла следует разместить команду `EXIT` или `EXIT WHEN`:

```
EXIT;  
EXIT WHEN условие;
```

Здесь *условие* — это логическое выражение.

В следующем примере команда `EXIT` прерывает выполнение цикла и передает управление команде, следующей за командой `END LOOP`. Функция `account_balance` возвращает остаток денег на счету с идентификатором `account_id`. Если на счету осталось менее 1000 долларов, выполняется команда `EXIT` и цикл завершается. В противном случае программа снимает с банковского счета клиента сумму, необходимую для оплаты заказов.

```
LOOP  
    balance_remaining := account_balance (account_id);  
    IF balance_remaining < 1000  
    THEN  
        EXIT;  
    ELSE  
        apply_balance (account_id, balance_remaining);  
    END IF;  
END LOOP;
```

Команда `EXIT` может использоваться только в цикле `LOOP`.

В PL/SQL для выхода из цикла также предусмотрена команда `EXIT WHEN`, предназначенная для завершения цикла с проверкой дополнительного условия. В сущности, `EXIT WHEN` сочетает в себе функции `IF-THEN` и `EXIT`. Приведенный пример можно переписать с использованием `EXIT WHEN`:

```
LOOP  
    /* Вычисление баланса */  
    balance_remaining := account_balance (account_id);  
  
    /* Условие встраивается в команду EXIT */  
    EXIT WHEN balance_remaining < 1000;  
    /* Если цикл все еще выполняется, с баланса списываются средства */  
    apply_balance (account_id, balance_remaining);  
END LOOP;
```

Как видите, во второй форме для проверки условия завершения команда `IF` не нужна. Логика проверки условия встраивается в команду `EXIT WHEN`. Так в каких же случаях следует использовать команду `EXIT WHEN`, а в каких — просто `EXIT`?

- Команда `EXIT WHEN` подойдет, когда условие завершения цикла определяется одним выражением. Предыдущий пример наглядно демонстрирует этот сценарий.
- При нескольких условиях завершения цикла или если при выходе должно быть определено возвращаемое значение, предпочтительнее `IF` или `CASE` с `EXIT`.

В следующем примере удобнее использовать команду EXIT. Фрагмент кода взят из функции, сравнивающей содержимое двух файлов:

```
...
IF (end_of_file1 AND end_of_file2)
THEN
    retval := TRUE;
    EXIT;
ELSIF (checkline != againstline)
THEN
    retval := FALSE;
    EXIT;
ELSIF (end_of_file1 OR end_of_file2)
THEN
    retval := FALSE;
    EXIT;
END IF;
END LOOP;
```

Моделирование цикла REPEAT UNTIL

В PL/SQL отсутствует традиционный цикл REPEAT UNTIL, в котором условие проверяется после выполнения тела цикла (что гарантирует выполнение тела как минимум один раз). Однако этот цикл легко моделируется простым циклом следующего вида:

```
LOOP
    ... тело цикла ...
    EXIT WHEN логическое_условие;
END LOOP;
```

Здесь *логическое_условие* — логическая переменная или выражение, результатом проверки которого является значение TRUE или FALSE (или NULL).

Бесконечный цикл

Некоторые программы (например, средства наблюдения за состоянием системы) рассчитаны на непрерывное выполнение с накоплением необходимой информации. В таких случаях можно намеренно использовать бесконечный цикл:

```
LOOP
    сбор_данных;
END LOOP;
```

Но каждый программист, имевший дело с заикливанием, подтвердит: бесконечный цикл обычно поглощает значительную часть ресурсов процессора. Проблема решается приостановкой выполнения между итерациями (и, разумеется, максимально возможной эффективностью сбора данных):

```
LOOP
    сбор_данных;
    DBMS_LOCK.sleep(10); -- ничего не делать в течение 10 секунд
END LOOP;
```

Во время приостановки программа практически не расходует ресурсы процессора.

Цикл WHILE

Условный цикл WHILE выполняется до тех пор, пока определенное в цикле условие остается равным TRUE. А поскольку возможность выполнения цикла зависит от условия и не ограничивается фиксированным количеством повторений, он используется именно в тех случаях, когда количество повторений цикла заранее не известно.

ПРЕРЫВАНИЕ БЕСКОНЕЧНОГО ЦИКЛА

На практике возможна ситуация, в которой бесконечный цикл потребуется завершить. Если цикл выполняется в анонимном блоке в SQL*Plus, скорее всего, проблему можно решить вводом терминальной комбинации завершения (обычно Ctrl+C). Но реальные программы чаще выполняются в виде сохраненных процедур, и даже уничтожение процесса, запустившего программу (например, SQL*Plus), не приведет к остановке фоновой задачи. Как насчет команды ALTER SYSTEM KILL SESSION? Хорошая идея, но в некоторых версиях Oracle эта команда не уничтожает заиклившись сеансы (почему — никто не знает). Как же «прикончить» выполняемую программу?

Возможно, вам придется прибегнуть к таким средствам операционной системы, как команда kill в Unix/Linux и orakill.exe в Microsoft Windows. Для выполнения этих команд необходимо знать идентификатор процесса «теневой задачи» Oracle; впрочем, нужную информацию легко получить при наличии привилегий чтения для представления V\$SESSION и V\$PROCESS. Но даже если неэлегантное решение вас не пугает, приходится учитывать другой фактор: в режиме сервера это, вероятно, приведет к уничтожению других сеансов. Лучшее решение, которое я могу предложить, — вставить в цикл своего рода «интерпретатор команд», использующий встроенный в базу данных механизм межпроцессных коммуникаций — «каналов» (pipes):

```
DECLARE
    pipename CONSTANT VARCHAR2(12) := 'signaler';
    result INTEGER;
    pipebuf VARCHAR2(64);
BEGIN
    /* Создание закрытого канала с известным именем */
    result := DBMS_PIPE.create_pipe(pipename);
    LOOP
        data_gathering_procedure;
        DBMS_LOCK.sleep(10);
        /* Проверка сообщений в канале */
        IF DBMS_PIPE.receive_message(pipename, 0) = 0
        THEN
            /* Интерпретация сообщения с соответствующими действиями */
            DBMS_PIPE.unpack_message(pipebuf);
            EXIT WHEN pipebuf = 'stop';
        END IF;
    END LOOP;
END;
```

Использование DBMS_PIPE не оказывает заметного влияния на общую загрузку процессора.

Простая вспомогательная программа может уничтожить заиклившуюся программу, отправив по каналу сообщение «stop»:

```
DECLARE
    pipename VARCHAR2 (12) := 'signaler';
    result INTEGER := DBMS_PIPE.create_pipe (pipename);
BEGIN
    DBMS_PIPE.pack_message ('stop');
    result := DBMS_PIPE.send_message (pipename);
END;
```

По каналу также можно отправлять другие команды — например, команду увеличения или уменьшения интервала ожидания. Кстати говоря, в приведенном примере используется закрытый канал, так что сообщение STOP должно отправляться с той же учетной записи пользователя, которая выполняет бесконечный цикл. Также следует заметить, что пространство имен базы данных для закрытых каналов глобально по отношению ко всем сеансам текущего пользователя. Следовательно, если вы захотите, чтобы в бесконечном цикле выполнялось сразу несколько программ, необходимо реализовать дополнительную логику для (1) создания имен каналов, уникальных для каждого сеанса, и (2) определения имен каналов для отправки команды STOP.

Общий синтаксис цикла **WHILE**:

```
WHILE условие
LOOP
    исполняемые_команды
END LOOP;
```

Здесь условие — логическая переменная или выражение, результатом проверки которого является логическое значение **TRUE**, **FALSE** или **NULL**. Условие проверяется при каждой итерации цикла. Если результат оказывается равным **TRUE**, тело цикла выполняется. Если же результат равен **FALSE** или **NULL**, то цикл завершается, а управление передается исполняемой команде, следующей за командой **END LOOP**. Основные свойства цикла **WHILE** приведены в таблице.

| Свойство | Описание |
|--|---|
| Условие завершения цикла | Если значением логического выражения цикла является FALSE или NULL |
| Когда проверяется условие завершения цикла | Перед первым и каждым последующим выполнением тела цикла. Таким образом, не гарантируется даже однократное выполнение тела цикла WHILE |
| В каких случаях используется данный цикл | (1) Если не известно, сколько раз будет выполняться тело цикла; (2) возможность выполнения цикла должна определяться условием; (3) тело цикла может не выполняться ни одного раза |

Условие **WHILE** проверяется в начале цикла и в начале каждой его итерации, перед выполнением тела цикла. Такого рода проверка имеет два важных последствия:

- Вся информация, необходимая для вычисления условия, должна задаваться перед первым выполнением цикла.
- Может оказаться, что цикл **WHILE** не будет выполнен ни одного раза.

Следующий пример цикла **WHILE** взят из файла **datemgr.pkg**, размещенного на сайте книги. Здесь используется условие, представленное сложным логическим выражением. Прерывание цикла **WHILE** вызвано одной из двух причин: либо завершением списка масок даты, которые применяются для выполнения преобразования, либо успешным завершением преобразования (и теперь переменная **date_converted** содержит значение **TRUE**):

```
/* Файл в Сети: datemgr.pkg */
WHILE mask_index <= mask_count AND NOT date_converted
LOOP
    BEGIN
        /* Попытка преобразования строки по маске в записи таблицы */
        retval := TO_DATE (value_in, fmts (mask_index));
        date_converted := TRUE;
    EXCEPTION
        WHEN OTHERS
        THEN
            mask_index:= mask_index+ 1;
    END;
END LOOP;
```

Цикл **FOR** со счетчиком

В **PL/SQL** существует два вида цикла **FOR**: с числовым счетчиком и с курсором. Цикл со счетчиком — это традиционный, хорошо знакомый всем программистам цикл **FOR**, поддерживаемый в большинстве языков программирования. Количество итераций этого цикла известно еще до его начала; оно задается в диапазоне между ключевыми словами **FOR** и **LOOP**.

Диапазон неявно объявляет управляющую переменную цикла (если она не была явно объявлена ранее), определяет начальное и конечное значения диапазона, а также задает направление изменения счетчика (по возрастанию или по убыванию).

Общий синтаксис цикла FOR:

```
FOR счетчик IN [REVERSE] начальное_значение .. конечное_значение
LOOP
    исполняемые_команды
END LOOP;
```

Между ключевыми словами LOOP и END LOOP должна стоять хотя бы одна исполняемая команда. Свойства цикла FOR с числовым счетчиком приведены в следующей таблице.

| Свойство | Описание |
|--|---|
| Условие завершения цикла | Числовой цикл FOR безусловно завершается при выполнении количества итераций, определенного диапазоном значений счетчика. (Цикл может завершаться и командой EXIT, но делать этого не рекомендуется) |
| Когда проверяется условие завершения цикла | После каждого выполнения тела цикла компилятор PL/SQL проверяет значение счетчика. Если оно выходит за пределы заданного диапазона, выполнение цикла прекращается. Если начальное значение больше конечного, то тело цикла не выполняется ни разу |
| В каких случаях используется данный цикл | Если тело цикла должно быть выполнено определенное количество раз, а выполнение не должно прерываться преждевременно |

Правила для циклов FOR с числовым счетчиком

При использовании цикла FOR с числовым счетчиком необходимо следовать некоторым правилам:

- **Не объявляйте счетчик цикла.** PL/SQL автоматически неявно объявляет локальную переменную с типом данных INTEGER. Область действия этой переменной совпадает с границей цикла; обращаться к счетчику за пределами цикла нельзя.
- **Выражения, используемые при определении диапазона (начального и конечного значений), вычисляются один раз.** Они не пересчитываются в ходе выполнения цикла. Если изменить внутри цикла переменные, используемые для определения диапазона значений счетчика, его границы останутся прежними.
- **Никогда не меняйте значения счетчика и границ диапазона внутри цикла.** Это в высшей степени порочная практика. Компилятор PL/SQL либо выдаст сообщение об ошибке, либо проигнорирует изменения — в любом случае возникнут проблемы.
- **Чтобы значения счетчика уменьшались в направлении от конечного к начальному, используйте ключевое слово REVERSE.** При этом первое значение в определении диапазона (*начальное_значение*) должно быть меньше второго (*конечное_значение*). Не меняйте порядок следования значений — просто поставьте ключевое слово REVERSE.

Примеры циклов FOR с числовым счетчиком

Следующие примеры демонстрируют некоторые варианты синтаксиса циклов FOR с числовым счетчиком.

- Цикл выполняется 10 раз; счетчик увеличивается от 1 до 10:

```
FOR loop_counter IN 1 .. 10
LOOP
    ... исполняемые_команды ...
END LOOP;
```

- Цикл выполняется 10 раз; счетчик уменьшается от 10 до 1:

```
FOR loop_counter IN REVERSE 1 .. 10
LOOP
    ... исполняемые_команды ...
END LOOP;
```

- Цикл не выполняется ни разу. В заголовке цикла указано ключевое слово **REVERSE**, поэтому счетчик цикла `loop_counter` изменяется от большего значения к меньшему. Однако начальное и конечное значения заданы в неверном порядке:

```
FOR loop_counter IN REVERSE 10 .. 1
LOOP
    /* Тело цикла не выполнится ни разу! */
    ...
END LOOP;
```

Даже если задать обратное направление с помощью ключевого слова **REVERSE**, меньшее значение счетчика все равно должно быть задано перед большим. Если первое число больше второго, тело цикла не будет выполнено. Если же граничные значения одинаковы, то тело цикла будет выполнено один раз.

- Цикл выполняется для диапазона, определяемого значениями переменной и выражения:

```
FOR calc_index IN start_period_number ..
    LEAST (end_period_number, current_period)
LOOP
    ... исполняемые команды ...
END LOOP;
```

В этом примере количество итераций цикла определяется во время выполнения программы. Начальное и конечное значения вычисляются один раз, перед началом цикла, и затем используются в течение всего времени его выполнения.

Нетривиальные приращения

В PL/SQL не предусмотрен синтаксис задания шага приращения счетчика. Во всех разновидностях цикла **FOR** с числовым счетчиком значение счетчика на каждой итерации всегда увеличивается или уменьшается на единицу.

Если приращение должно быть отлично от единицы, придется писать специальный код. Например, что нужно сделать, чтобы тело цикла выполнялось только для четных чисел из диапазона от 1 до 100? Во-первых, можно использовать числовую функцию **MOD**, как в следующем примере:

```
FOR loop_index IN 1 .. 100
LOOP
    IF MOD (loop_index, 2) = 0
    THEN
        /* Число четное, поэтому вычисления выполняются */
        calc_values (loop_index);
    END IF;
END LOOP;
```

Также возможен и другой способ — умножить значение счетчика на два и использовать вдвое меньший диапазон:

```
FOR even_number IN 1 .. 50
LOOP
    calc_values (even_number*2);
END LOOP;
```

В обоих случаях процедура `calc_values` выполняется только для четных чисел. В первом примере цикл **FOR** повторяется 100 раз, во втором — только 50.

Но какое бы решение вы ни выбрали, обязательно подробно его опишите. Комментарии помогут другим программистам при сопровождении вашей программы.

Цикл FOR с курсором

Курсорная форма цикла FOR связывается с явно заданным курсором (а по сути, определяется им) или инструкцией SELECT, заданной непосредственно в границах цикла. Используйте эту форму только в том случае, если вам нужно извлечь и обработать все записи курсора (впрочем, при работе с курсорами это приходится делать довольно часто).

Цикл FOR с курсором — одна из замечательных возможностей PL/SQL, обеспечивающая тесную и эффективную интеграцию процедурных конструкций с мощностью языка доступа к базам данных SQL. Его применение заметно сокращает объем кода, необходимого для выборки данных из курсора, а также уменьшает вероятность возникновения ошибок при циклической обработке данных — ведь именно циклы являются одним из основных источников ошибок в программах.

Базовый синтаксис цикла FOR с курсором:

```
FOR запись IN { имя_курсора | (команда_SELECT) }
LOOP
    исполняемые команды
END LOOP;
```

Здесь *запись* — неявно объявленная запись с атрибутом %ROWTYPE для курсора *имя_курсора*.



Не объявляйте явно запись с таким же именем, как у индексной записи цикла. В этом нет необходимости, поскольку запись объявляется автоматически, к тому же это может привести к логическим ошибкам. О том, как получить доступ к информации о записях, обработанных в цикле FOR после его выполнения, рассказывается далее в этой главе.

В цикле FOR можно также задать не курсор, а непосредственно SQL-инструкцию SELECT, как показано в следующем примере:

```
FOR book_rec IN (SELECT * FROM books)
LOOP
    show_usage (book_rec);
END LOOP;
```

Мы не рекомендуем использовать эту форму, поскольку встраивание инструкций SELECT в «неожиданные» места кода затрудняет его сопровождение и отладку.

Свойства цикла FOR с использованием курсора приведены в следующей таблице.

| Свойство | Описание |
|--|---|
| Условие завершения цикла | Выборка всех записей курсора. Цикл можно завершить и командой EXIT, но поступать так не рекомендуется |
| Когда проверяется условие завершения цикла | После каждого выполнения тела цикла компилятор PL/SQL осуществляет выборку очередной записи. Если значение атрибута курсора %NOTFOUND% оказывается равным TRUE, цикл завершается. Если курсор не возвратит ни одной строки, тело цикла никогда не будет выполнено |
| В каких случаях используется данный цикл | При необходимости выбрать и обработать каждую запись курсора |

Примеры цикла FOR с курсором

Допустим, нам необходимо обновить счета владельцев всех животных, живущих в специальном отеле. Следующий пример включает анонимный блок, в котором для выбора номера комнаты и идентификатора животного используется курсор `occupancy_cur`. Процедура `update_bill` вносит все изменения в счет:

```
1  DECLARE
2      CURSOR occupancy_cur IS
3          SELECT pet_id, room_number
4              FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
5      occupancy_rec occupancy_cur%ROWTYPE;
6  BEGIN
7      OPEN occupancy_cur;
8      LOOP
9          FETCH occupancy_cur INTO occupancy_rec;
10         EXIT WHEN occupancy_cur%NOTFOUND;
11         update_bill
12             (occupancy_rec.pet_id, occupancy_rec.room_number);
13     END LOOP;
14     CLOSE occupancy_cur;
15 END;
```

Этот код последовательно и явно выполняет все необходимые действия: мы определяем курсор (строка 2), явно объявляем запись для этого курсора (строка 5), открываем курсор (строка 7), начинаем бесконечный цикл (строка 8), производим выборку записи из курсора (строка 9), проверяем условие выхода из цикла (конец данных) по атрибуту `%NOTFOUND` курсора (строка 10) и, наконец, выполняем обновление (строка 11). После этого программист должен закрыть курсор (строка 14).

Вот что получится, если переписать тот же код с использованием цикла `FOR` с курсором:

```
DECLARE
    CURSOR occupancy_cur IS
        SELECT pet_id, room_number
            FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
BEGIN
    FOR occupancy_rec IN occupancy_cur
    LOOP
        update_bill (occupancy_rec.pet_id, occupancy_rec.room_number);
    END LOOP;
END;
```

Как все просто и понятно! Исчезло объявление записи. Исчезли команды `OPEN`, `FETCH` и `CLOSE`. Больше не нужно проверять атрибут `%NOTFOUND`. Нет никаких сложностей с организацией выборки данных. По сути, вы говорите PL/SQL: «Мне нужна каждая строка таблицы, и я хочу, чтобы она была помещена в запись, соответствующую курсору». И PL/SQL делает то, что вы хотите, как это должен делать любой современный язык программирования.

Курсору в цикле `FOR`, как и любому другому курсору, можно передавать параметры. Если какой-либо из столбцов списка `SELECT` определяется выражением, обязательно определите для него псевдоним. Для обращения к конкретному значению в записи курсора в пределах цикла необходимо использовать «точечный» синтаксис (*имя_записи.имя_столбца* — например, `occupancy_rec.room_number`), так что без псевдонима к столбцу-выражению обратиться не удастся.

За дополнительной информацией о работе с курсорами обращайтесь к главе 15.

Метки циклов

Циклу можно присвоить имя при помощи метки (см. главу 3). Метка цикла в PL/SQL имеет стандартный формат:

```
<<имя_метки>>
```

Метка располагается непосредственно перед командой LOOP:

```
<<all_emps>>
FOR emp_rec IN emp_cur
LOOP
```

```
...
END LOOP;
```

Эту же метку можно указать и после ключевых слов END LOOP, как в следующем примере:

```
<<year_loop>>
WHILE year_number <= 1995
LOOP
  <<month_loop>>
  FOR month_number IN 1 .. 12
  LOOP
    ...
  END LOOP month_loop;
  year_number := year_number + 1;
```

```
END LOOP year_loop;
```

Метки циклов могут пригодиться в нескольких типичных ситуациях:

- Если вы написали очень длинный цикл с множеством вложенных циклов (допустим, начинающийся в строке 50, завершается в строке 725 и содержащий 16 вложенных циклов), используйте метку цикла для того, чтобы явно связать его конец с началом. Визуальная пометка поможет при отладке и сопровождении программы; без нее будет трудно уследить, какая команда LOOP соответствует каждой из команд END LOOP.
- Метку цикла можно использовать для уточнения имени управляющей переменной цикла (записи или счетчика), что также упрощает чтение программы:

```
<<year_loop>>
FOR year_number IN 1800..1995
LOOP
  <<month_loop>>
  FOR month_number IN 1 .. 12
  LOOP
    IF year_loop.year_number = 1900 THEN ... END IF;
  END LOOP month_loop;
END LOOP year_loop;
```

- При использовании вложенных циклов метки упрощают чтение кода и повышают эффективность их выполнения. При желании выполнение именованного внешнего цикла можно прервать при помощи команды EXIT с заданной в нем меткой цикла:

```
EXIT метка_цикла;
EXIT метка_цикла WHEN условие;
```

Но обычно применять метки циклов подобным образом не рекомендуется, так как они ухудшают структуру логики программы (по аналогии с GOTO) и усложняют отладку. Если вам потребуется использовать подобный код, лучше изменить структуру цикла, а возможно, заменить его простым циклом или WHILE.

Команда CONTINUE

В Oracle11g появилась новая возможность для работы с циклами: команда CONTINUE. Он используется для выхода из текущей итерации цикла и немедленного перехода

к следующей итерации. Как и EXIT, эта команда существует в двух формах: безусловной (CONTINUE) и условной (CONTINUE WHEN).

Простой пример использования CONTINUE WHEN для пропуска итераций с четными значениями счетчика:

```
BEGIN
  FOR l_index IN 1 .. 10
  LOOP
    CONTINUE WHEN MOD (l_index, 2) = 0;
    DBMS_OUTPUT.PUT_LINE ('Счетчик = ' || TO_CHAR (l_index));
  END LOOP;
END;
```

Результат:

```
Счетчик = 1
Счетчик = 3
Счетчик = 5
Счетчик = 7
Счетчик = 9
```

Конечно, того же эффекта можно добиться при помощи команды IF, но команда CONTINUE предоставляет более элегантный и понятный способ представления реализуемой логики.

Команда CONTINUE чаще всего применяется для модификации существующего кода с внесением целенаправленных изменений и немедленным выходом из цикла для предотвращения побочных эффектов.

ТАК ЛИ ПЛОХА КОМАНДА CONTINUE?

Когда я впервые узнал о команде CONTINUE, на первый взгляд мне показалось, что она представляет очередную форму неструктурированной передачи управления по аналогии с GOTO, поэтому ее следует по возможности избегать (я прекрасно обходился без нее годами!). Чарльз Уэзерелл, один из руководителей группы разработки PL/SQL, развеял мои заблуждения:

Уже давно (еще в эпоху знаменитого манифеста Дейкстры «о вреде goto») конструкции exit и continue были проанализированы и отнесены к структурным средствам передачи управления. Более того, команда exit была признана в одной из авторитетных работ Кнута как способ корректного прерывания вычислений.

Бем и Якопини доказали, что любая программа, использующая произвольные синхронные управляющие элементы (например, циклы или goto), может быть переписана с использованием циклов while, команд if и логических переменных в полностью структурной форме. Более того, преобразование между «плохой» неструктурированной версией и «хорошей» структурированной версией в программе может быть автоматизировано. К сожалению, новая «хорошая» программа может на порядок увеличиваться в размерах из-за необходимости введения многочисленных логических переменных и копирования кода во множественные ветви if. На практике в реальных программах такое увеличение размера встречается редко, но для моделирования эффекта continue и exit часто применяется копирование кода. Оно создает проблемы с сопровождением, потому что если в будущем программу потребуется модифицировать, программист должен помнить, что изменить нужно все копии вставленного кода.

Команда continue полезна тем, что она делает код более компактным и понятным, а также сокращает необходимость в логических переменных, смысл которых трудно

понять с первого взгляда. Чаще всего она используется в циклах, в которых точная обработка каждого элемента зависит от подробных структурных тестов. Заготовка цикла может выглядеть так, как показано ниже; обратите внимание на команду exit, которая проверяет, не пора ли завершить обработку. Также стоит заметить, что последняя команда continue (после условия5) не является строго необходимой. С другой стороны, включение continue после каждого действия упрощает добавление новых действий в произвольном порядке без нарушения работоспособности других действий.

```
LOOP
  EXIT WHEN условие_выхода;
  CONTINUE WHEN условие1;
  CONTINUE WHEN условие2;
  подготовительная_фаза;
  IF условие4 THEN
    выполнено_действие4;
    CONTINUE;
  END IF;
  IF условие5 THEN
    выполнено_действие5;
    CONTINUE; -- Не является строго необходимой.
  END IF;
END LOOP;
```

Без команды continue мне пришлось бы реализовать тело цикла следующим образом:

```
LOOP
  EXIT WHEN exit_condition_met;
  IF condition1
    THEN
      NULL;
    ELSIF condition2
    THEN
      NULL;
    ELSE
      setup_steps_here;
      IF condition4 THEN
        action4_executed;
      ELSIF condition5 THEN
        action5_executed;
      END IF;
    END IF;
  END LOOP;
```

Даже в этом простом примере команда continue позволяет обойтись без нескольких секций elsif, сокращает уровень вложенности и наглядно показывает, какие логические проверки (и сопутствующая обработка) должны выполняться на том же уровне. В частности, continue существенно сокращает глубину вложенности. Умение правильно использовать команду continue безусловно помогает программистам PL/SQL писать более качественный код.

Также команда CONTINUE пригодится для завершения внутренних циклов и немедленного продолжения следующей итерации внешнего цикла. Для этого циклам присваиваются имена при помощи меток. Пример:

```
BEGIN
  <<outer>>
  FOR outer_index IN 1 .. 5
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Внешний счетчик = ' || TO_CHAR (outer_index));

    <<inner>>
    FOR inner_index IN 1 .. 5
```

```

        LOOP
            DBMS_OUTPUT.PUT_LINE (
                '    Внутренний счетчик = ' || TO_CHAR (inner_index));
            CONTINUE outer;
        END LOOP inner;
    END LOOP outer;
END;
/

```

Результат:

```

Внешний счетчик = 1
    Внутренний счетчик = 1
Внешний счетчик = 2
    Внутренний счетчик = 1
Внешний счетчик = 3
    Внутренний счетчик = 1
Внешний счетчик = 4
    Внутренний счетчик = 1
Внешний счетчик = 5
    Внутренний счетчик = 1

```

Полезные советы

Циклы — очень мощные и полезные конструкции, но при их использовании необходима осторожность. Именно циклы часто создают проблемы с быстродействием программ, и любая ошибка, возникшая в цикле, повторяется ввиду многократности его выполнения. Логика, определяющая условие остановки цикла, бывает очень сложной. В этом разделе приводятся несколько советов по поводу того, как сделать циклы более четкими и понятными, а также упростить их сопровождение.

Используйте понятные имена для счетчиков циклов

Не заставляйте программиста, которому поручено сопровождать программу, с помощью сложной дедукции определять смысл начального и конечного значения счетчика цикла FOR. Применяйте понятные и информативные имена переменных и циклов, и тогда другим программистам (да и вам самим некоторое время спустя) легко будет разобраться в таком коде.

Как можно понять следующий ход, не говоря уже о его сопровождении?

```

FOR i IN start_id .. end_id
LOOP
    FOR j IN 1 .. 7
    LOOP
        FOR k IN 1 .. 24
        LOOP
            build_schedule (i, j, k);
        END LOOP;
    END LOOP;
END LOOP;

```

Трудно представить, зачем использовать однобуквенные имена переменных, словно сошедшие со страниц учебника начального курса алгебры, но это происходит сплошь и рядом. Вредные привычки, приобретенные на заре эпохи программирования, искоренить невероятно сложно. А исправить этот код очень просто — достаточно присвоить переменным более информативные имена:

```

FOR focus_account IN start_id .. end_id
LOOP
    FOR day_in_week IN 1 .. 7

```

```

LOOP
  FOR month_in_biyear IN 1 .. 24
    LOOP
      build_schedule (focus_account, day_in_week, month_in_biyear);
    END LOOP;
  END LOOP;
END LOOP;

```

С содержательными именами переменных сразу видно, что внутренний цикл просто перебирает месяцы двухлетнего периода ($12 \times 2 = 24$).

Корректно выходите из цикла

Один из фундаментальных принципов структурного программирования звучит так: «один вход, один выход»; иначе говоря, программа должна иметь одну точку входа и одну точку выхода. Первая часть в PL/SQL реализуется автоматически. Какой бы цикл вы ни выбрали, у него всегда только одна точка входа — первая исполняемая команда, следующая за ключевым словом `LOOP`. Но вполне реально написать цикл с несколькими точками выхода. Однако так поступать не рекомендуется, поскольку цикл с несколькими путями выхода трудно отлаживать и сопровождать.

При завершении цикла следует придерживаться следующих правил:

- Не используйте в циклах `FOR` и `WHILE` команды `EXIT` и `EXIT WHEN`. Цикл `FOR` должен завершаться только тогда, когда исчерпаны все значения диапазона (целые числа или записи). Команда `EXIT` в цикле `FOR` прерывает этот процесс, а следовательно, идет вразрез с самим назначением цикла `FOR`. Точно так же условие окончания цикла `WHILE` задается в самой команде `WHILE` и нигде более задавать или дополнять его не следует.
- Не используйте в циклах команды `RETURN` и `GOTO`, поскольку это вызывает преждевременное и неструктурированное завершение цикла. Применение указанных команд может выглядеть заманчиво, так как они сокращают объем кода. Однако спустя некоторое время вы потратите больше времени, пытаясь понять, изменить и отладить такой код.

Рассмотрим суть этих правил на примере цикла `FOR` с курсором. Как вы уже видели, данный цикл облегчает перебор возвращаемых курсором записей, но не подходит для случаев, когда выход из цикла определяется некоторым условием, основанным на данных текущей записи. Предположим, что в цикле записи курсора просматриваются до тех пор, пока сумма значений определенного столбца не превысит максимальное значение, как показано в следующем примере. Хотя это можно сделать с помощью цикла `FOR` и курсора, выполнив внутри этого цикла команду `EXIT`, поступать так не следует.

```

1  DECLARE
2      CURSOR occupancy_cur IS
3          SELECT pet_id, room_number
4              FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
5      pet_count INTEGER := 0;
6  BEGIN
7      FOR occupancy_rec IN occupancy_cur
8          LOOP
9              update_bill
10                 (occupancy_rec.pet_id, occupancy_rec.room_number);
11              pet_count := pet_count + 1;
12              EXIT WHEN pet_count >= pets_global.max_pets;
13          END LOOP;
14  END;

```

В заголовке цикла `FOR` явно указано, что его тело должно быть выполнено n раз (где n — количество итераций в цикле со счетчиком или количество записей в цикле с курсором). Команда `EXIT` в цикле `FOR` (строка 12) изменяет логику его выполнения, и в результате получается код, который трудно понять и отладить.

Поэтому если нужно прервать цикл на основании информации текущей записи, лучше воспользоваться циклом `WHILE` или простым циклом, чтобы структура кода лучше отражала ваши намерения.

Получение информации о выполнении цикла `FOR`

Циклы `FOR` — удобные, четко формализованные структуры, которые выполняют в программе большую «административную» работу (особенно циклы с курсором). Однако у них есть и существенный недостаток: позволяя Oracle выполнять работу за нас, мы ограничиваем собственные возможности доступа к конечным результатам цикла после его завершения.

Предположим, нам нужно узнать, сколько записей обработано в цикле `FOR` с курсором, и затем использовать это значение в программе. Было бы очень удобно написать примерно такой код:

```
BEGIN
  FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
  LOOP
    ... обработка данных ...
  END LOOP;
  IF books_cur%ROWCOUNT > 10 THEN ...
```

Но попытавшись это сделать, мы получим сообщение об ошибке, поскольку курсор неявно открывается и закрывается Oracle. Как же получить нужную информацию из уже завершившегося цикла? Для этого следует объявить переменную в том блоке, в который входит цикл `FOR`, и присвоить ей значение в теле цикла — в таком случае переменная останется доступной и после завершения цикла. Вот как это делается:

```
DECLARE
  book_count PLS_INTEGER := 0;
BEGIN
  FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
  LOOP
    ... обработка данных ...
    book_count := books_cur%ROWCOUNT;
  END LOOP;
  IF book_count > 10 THEN ...
```

Команда `SQL` как цикл

На самом деле команда `SQL` (например, `SELECT`) тоже может рассматриваться как цикл, потому что она определяет действие, выполняемое компилятором `SQL` с набором данных. В некоторых случаях при реализации определенной задачи можно даже выбрать между использованием цикла `PL/SQL` и команды `SQL`. Давайте рассмотрим пример, а затем сделаем некоторые выводы о том, какое решение лучше.

Предположим, наша программа должна перенести информацию о выбывших из отеля животных из таблицы `occupancy` в таблицу `occupancy_history`. Опытный программист `PL/SQL` сходу выбирает цикл `FOR` с курсором. В теле цикла каждая выбранная из курсора запись сначала добавляется в таблицу `occupancy_history`, а затем удаляется из таблицы `occupancy`:

```
DECLARE
  CURSOR checked_out_cur IS
    SELECT pet_id, name, checkout_date
    FROM occupancy WHERE checkout_date IS NOT NULL;
BEGIN
  FOR checked_out_rec IN checked_out_cur
  LOOP
    INSERT INTO occupancy_history (pet_id, name, checkout_date)
    VALUES (checked_out_rec.pet_id, checked_out_rec.name,
```

```

        checked_out_rec.checkout_date);
DELETE FROM occupancy WHERE pet_id = checked_out_rec.pet_id;
END LOOP;
END;
```

Программа работает, но является ли данное решение единственным? Конечно же, нет. Ту же логику можно реализовать с помощью команд SQL INSERT-SELECT FROM с последующей командой DELETE:

```

BEGIN
  INSERT INTO occupancy_history (pet_id, NAME, checkout_date)
    SELECT pet_id, NAME, checkout_date
      FROM occupancy WHERE checkout_date IS NOT NULL;
  DELETE FROM occupancy WHERE checkout_date IS NOT NULL;
END;
```

Каковы преимущества такого подхода? Код стал короче и выполняется более эффективно благодаря уменьшению количества переключений контекста (переходов от исполняемого ядра PL/SQL к исполняемому ядру SQL и обратно). Теперь обрабатываются только одна команда INSERT и одна команда DELETE.

Однако у «чистого» SQL-подхода имеются свои недостатки. Команда SQL обычно действует по принципу «все или ничего». Иначе говоря, если при обработке хотя бы одной записи occupancy_history происходит ошибка, то отменяются все инструкции INSERT и DELETE и ни одна запись не будет вставлена или удалена. Кроме того, приходится дважды записывать условие WHERE. В данном примере это не очень важно, но в более сложных запросах данное обстоятельство может иметь решающее значение. А первоначальный цикл FOR позволяет избежать дублирования потенциально сложной логики в нескольких местах.

Кроме того, PL/SQL превосходит SQL в отношении гибкости. Допустим, нам хотелось бы переносить за одну операцию максимально возможное количество записей, а для тех записей, при перемещении которых произошли ошибки, просто записывать сообщения в журнал. В этом случае стоит воспользоваться циклом FOR с курсором, дополненным разделом исключений:

```

BEGIN
  FOR checked_out_rec IN checked_out_cur
  LOOP
    BEGIN
      INSERT INTO occupancy_history ...
      DELETE FROM occupancy ...
    EXCEPTION
      WHEN OTHERS THEN
        log_checkout_error (checked_out_rec);
    END;
  END LOOP;
END;
```

PL/SQL позволяет обрабатывать записи по одной и для каждой из них выполнять необходимые действия (которые могут базироваться на сложной процедурной логике, зависящей от содержимого конкретной записи). В таких случаях удобнее использовать комбинацию PL/SQL и SQL. Но если ваша задача позволяет ограничиться одним SQL, лучше им и воспользоваться — код получится и короче, и эффективнее.



Продолжить выполнение после ошибок в командах SQL можно двумя способами: (1) использовать конструкцию LOG ERRORS со вставкой, обновлением и удалением в Oracle10g Release 2 и выше; и (2) использовать конструкцию SAVE EXCEPTIONS в командах FOR ALL. За дополнительной информацией обращайтесь к главе 21.

6

Обработка исключений

К сожалению, многие программисты не склонны тратить время на то, чтобы застраховать свой код от всех возможных неожиданностей. У большинства из нас хватает проблем с написанием кода, реализующего положительные аспекты приложения: управление данными клиентов, построение счетов и т. д.; вдобавок это увеличивает объем работы. Всегда бывает дьявольски сложно — как с психологической точки зрения, так и в отношении расходования ресурсов — сосредоточиться на негативных аспектах работы системы: что, если пользователь нажмет не ту клавишу? А что делать, если база данных недоступна?

В результате мы пишем приложения, предназначенные для работы в «идеальном мире», где в программах не бывает ошибок, пользователи вводят лишь правильные данные, а все системы — и аппаратные и программные — всегда в полном порядке.

Конечно, жестокая реальность устанавливает свои правила: как бы вы ни старались, в приложении все равно отыщется еще одна ошибка. А ваши пользователи всегда постараются отыскать последовательность нажатий клавиш, от которых форма перестанет работать. Проблема проста: либо вы выделяете время на отладку и защиту своих программ, либо вам придется вести бесконечные бои в отступление, принимая отчаянные звонки от пользователей и пытаясь потушить разгорающееся пламя.

К счастью, PL/SQL предоставляет достаточно мощный и гибкий механизм перехвата и обработки ошибок. И вполне возможно написать на языке PL/SQL такое приложение, которое полностью защитит от ошибок и всех пользователей, и базу данных.

Основные концепции и терминология обработки исключений

В языке PL/SQL ошибки всех видов интерпретируются как *исключения* — ситуации, которые не должны возникать при нормальном выполнении программы.

К числу исключений относятся:

- ошибки, генерируемые системой (например, нехватка памяти или повторяющееся значение индекса);
- ошибки, вызванные действиями пользователя;
- предупреждения, выдаваемые приложением пользователю.

PL/SQL перехватывает ошибки и реагирует на них при помощи так называемых обработчиков исключений. Механизм обработчиков исключений позволяет четко отделить

код обработки ошибок от основной логики программы, а также дает возможность реализовать обработку ошибок, *управляемую событиями* (в отличие от старой линейной модели). Независимо от того, как и по какой причине возникло конкретное исключение, оно всегда обрабатывается одним и тем же обработчиком в разделе исключений.

При возникновении ошибки — как системной, так и ошибки в приложении — в PL/SQL инициируется исключение. В результате выполнение блока прерывается, и управление передается для обработки в раздел исключений текущего блока, если он имеется. После обработки исключения возврат в тот блок, где исключение было инициировано, невозможен, поэтому управление передается во внешний блок.

Схема передачи управления при возникновении исключения показана на рис. 6.1.

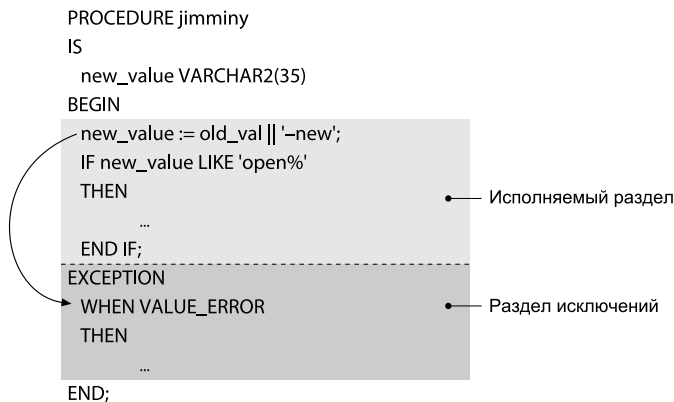


Рис. 6.1. Архитектура обработки исключений

Существует два типа исключений:

- **Системное исключение** определяется в Oracle и обычно инициируется исполняемым ядром PL/SQL, обнаружившим ошибку. Одним системным исключениям присваиваются имена (например, `NO_DATA_FOUND`), другие ограничиваются номерами и описаниями.
- **Исключение, определяемое программистом**, актуально только для конкретного приложения. Имя исключения можно связать с конкретной ошибкой Oracle с помощью директивы компилятора `EXCEPTION_INIT` или же назначить ошибке номер и описание процедурой `RAISE_APPLICATION_ERROR`.

В этой главе будут использоваться следующие термины:

- **Раздел исключений** — необязательный раздел блока PL/SQL (анонимного блока, процедуры, функции, триггера или инициализационного раздела пакета), содержащий один или несколько обработчиков исключений. Структура раздела исключений очень похожа на структуру команды `CASE`, о которой рассказывалось в главе 4.
- **Инициировать исключение** — значит остановить выполнение текущего блока PL/SQL, оповещая исполняемое ядро об ошибке. Исключение может инициировать либо Oracle, либо ваш собственный программный код при помощи команды `RAISE` или процедуры `RAISE_APPLICATION_ERROR`.
- **Обработать исключение** — значит перехватить ошибку, передав управление обработчику исключения. Написанный программистом обработчик может содержать код, который в ответ на исключение выполняет определенные действия (например, записывает информацию об ошибке в журнал, выводит сообщение для пользователя или передает исключение во внешний блок).

- **Область действия** — часть кода (конкретный блок или весь раздел), в котором может инициироваться исключение, а также часть кода, инициируемые исключения которого могут перехватываться и обрабатываться соответствующим разделом исключений.
- **Передача исключения** — процесс передачи исключения во внешний блок, если в текущем блоке это исключение не обработано.
- **Необработанное исключение** — исключение, которое передается без обработки из «самого внешнего» блока PL/SQL. После этого управление передается исполнительной среде, которая уже сама определяет, как отреагировать на исключение (выполнить откат транзакции, вывести сообщение об ошибке, проигнорировать ее и т. д.).
- **Анонимное исключение** — исключение, с которым связан код ошибки и описание. Такое исключение не имеет имени, которое можно было бы использовать в команде RAISE или секции WHEN обработчика исключений.
- **Именованное исключение** — исключение, которому имя присвоено либо Oracle (в одном из встроенных пакетов), либо разработчиком. В частности, для этой цели можно использовать директиву компилятора EXCEPTION_INIT (в таком случае имя можно будет применять и для инициирования, и для обработки исключения).

Определение исключений

Прежде чем исключение можно будет инициировать и обрабатывать, его необходимо определить. В Oracle заранее определены тысячи исключений, большинство из которых имеют только номера и пояснительные сообщения. Имена присваиваются только самым распространенным исключениям.

Имена присваиваются в пакете STANDARD (одном из двух пакетов по умолчанию PL/SQL; другой пакет — DBMS_STANDARD), а также в других встроенных пакетах, таких как UTL_FILE и DBMS_SQL. Код, используемый Oracle для определения исключений (таких, как NO_DATA_FOUND), не отличается от кода, который вы будете использовать для определения или объявления ваших собственных исключений.

Это можно сделать двумя способами, описанными ниже.

Объявление именованных исключений

Исключения PL/SQL, объявленные в пакете STANDARD и в других встроенных пакетах, представляют внутренние (то есть системные) ошибки. Однако многие проблемы, с которыми будет сталкиваться пользователь приложения, актуальны только в этом конкретном приложении. Возможно, вашей программе придется перехватывать и обрабатывать такие ошибки, как «отрицательный баланс счета» или «дата обращения не может быть меньше текущей даты». Хотя эти ошибки имеют иную природу, нежели, скажем, ошибки «деления на ноль», они также относятся к разряду исключений, связанных с нормальной работой программы, и должны обрабатываться этой программой.

Одной из самых полезных особенностей обработки исключений PL/SQL является отсутствие структурных различий между внутренними ошибками и ошибками конкретных приложений. Любое исключение может и должно обрабатываться в разделе исключений независимо от типа ошибки.

Конечно, для обработки исключения необходимо знать его имя. Поскольку в PL/SQL имена пользовательским исключениям автоматически не назначаются, вы должны делать это самостоятельно, определяя исключения в разделе объявлений блока PL/SQL. При этом задается имя исключения, за которым следует ключевое слово EXCEPTION:

```
имя_исключения EXCEPTION;
```

Следующий раздел объявлений процедуры `calc_annual_sales` содержит два объявления исключений, определяемых программистом:

```
PROCEDURE calc_annual_sales
  (company_id_in IN company.company_id%TYPE)
IS
  invalid_company_id  EXCEPTION;
  negative_balance    EXCEPTION;

  duplicate_company   BOOLEAN;
BEGIN
  ... исполняемые команды ...
EXCEPTION
  WHEN NO_DATA_FOUND  -- системное исключение
  THEN
    ...
  WHEN invalid_company_id
  THEN

  WHEN negative_balance
  THEN
    ...
END;
```

По своему формату имена исключений схожи с именами других переменных, но ссылаться на них можно только двумя способами:

- В команде `RAISE`, находящейся в исполняемом разделе программы (для инициирования исключения):
`RAISE invalid_company_id;`
- В секции `WHEN` раздела исключений (для обработки инициированного исключения):
`WHEN invalid_company_id THEN`

Связывание имени исключения с кодом ошибки

В Oracle, как уже было сказано, имена определены лишь для самых распространенных исключений. Тысячи других ошибок в СУБД имеют лишь номера и снабжены пояснительными сообщениями. Вдобавок инициировать исключение с номером ошибки (в диапазоне от `-20 999` до `-20 000`) может и разработчик приложения, воспользовавшись для этой цели процедурой `RAISE_APPLICATION_ERROR` (см. далее раздел «Инициирование исключений»).

Наличие в программном коде исключений без имен вполне допустимо, но такой код малопонятен и его трудно сопровождать. Допустим, вы написали программу, при выполнении которой Oracle выдает ошибку, связанную с данными, например `ORA-01843: not a valid month`. Для перехвата этой ошибки в программу включается обработчик следующего вида:

```
EXCEPTION
  WHEN OTHERS THEN
    IF SQLCODE = -1843 THEN
```

Но код получается совершенно непонятным. Чтобы сделать смысл этого кода более очевидным, следует воспользоваться директивой `EXCEPTION_INIT`.



Встроенная функция `SQLCODE` возвращает номер последней сгенерированной ошибки. Она будет рассмотрена далее в разделе «Обработка исключений» этой главы.

Директива EXCEPTION_INIT

Директива компилятора EXCEPTION_INIT (команда, выполняемая во время компиляции) связывает идентификатор, объявленный с ключевым словом EXCEPTION, с внутренним кодом ошибки. Установив такую связь, можно инициировать исключение по имени и указать это имя в условии WHEN обработчика ошибок.

С директивой EXCEPTION_INIT условие WHEN, использованное в предыдущем примере, приводится к следующему виду:

```
PROCEDURE my_procedure
IS
    invalid_month EXCEPTION;
    PRAGMA EXCEPTION_INIT (invalid_month, -1843);
BEGIN
    ...
EXCEPTION
    WHEN invalid_month THEN
```

Жесткое кодирование номера ошибки становится излишним; имя ошибки говорит само за себя.

Директива EXCEPTION_INIT должна располагаться в разделе объявлений блока. Указанное в ней исключение должно быть объявлено либо в том же блоке, либо во внешнем, либо в спецификации пакета. Синтаксис директивы в анонимном блоке:

```
DECLARE
    имя_исключения EXCEPTION;
    PRAGMA EXCEPTION_INIT (имя_исключения, целое_число);
```

Здесь *имя_исключения* — имя исключения, объявляемого программистом, а *целое_число* — номер ошибки Oracle, которую следует связать с данным исключением. Номером ошибки может служить любое число со следующими ограничениями:

- Номер ошибки не может быть равен -1403 (один из двух кодов ошибок NO_DATA_FOUND). Если вы по какой-либо причине захотите связать свое именованное исключение с этой ошибкой, передайте директиве EXCEPTION_INIT значение 100.
- Номер ошибки не может быть равен 0 или любому положительному числу, кроме 100.
- Номер ошибки не может быть отрицательным числом, меньшим -1 000 000.

Рассмотрим пример возможного объявления исключения. В приведенном ниже программном коде я объявляю и связываю исключение со следующим номером:

```
ORA-2292 integrity constraint (OWNER.CONSTRAINT) violated -
    child record found.
```

Ошибка происходит при попытке удаления родительской записи, у которой в таблице имеются *дочерние записи* (то есть записи с внешним ключом, ссылающимся на родительскую запись):

```
PROCEDURE delete_company (company_id_in IN NUMBER)
IS
    /* Объявление исключения. */
    still_have_employees EXCEPTION;

    /* Имя исключения связывается с номером ошибки. */
    PRAGMA EXCEPTION_INIT (still_have_employees, 2292);
BEGIN
    /* Попытка удаления информации о компании. */
    DELETE FROM company
        WHERE company_id = company_id_in;
EXCEPTION
    /* При обнаружении дочерних записей иницируется это исключение! */
    WHEN still_have_employees
```

```

THEN
    DBMS_OUTPUT.PUT_LINE
        ('Пожалуйста, сначала удалите данные о служащих компании.');
```

END;

Рекомендации по использованию EXCEPTION_INIT

Директиву EXCEPTION_INIT целесообразно использовать в двух ситуациях:

- при необходимости присвоить имя безымянному системному исключению, задействованному в программе (следовательно, если в Oracle не определено имя для некоторой ошибки, это еще не означает, что с ней можно работать только по номеру);
- когда нужно присвоить имя специфическому для приложения исключению, иницирующему процедурой RAISE_APPLICATION_ERROR (см. далее раздел «Инициирование исключений»). Это позволяет обрабатывать данное исключение по имени, а не по номеру.

В обоих случаях все директивы EXCEPTION_INIT желательно объединить в пакет, чтобы определения исключений не были разбросаны по всему коду приложения. Допустим, вы интенсивно используете динамический SQL (см. главу 16), и при выполнении запросов часто возникает ошибка «invalid column name» (неверное имя столбца). Запоминать код ошибки не хочется, но и определять директивы имя для исключения в 20 разных программах тоже неразумно. Поэтому имеет смысл определить собственные «системные исключения» в отдельном пакете для работы с динамическим SQL:

```

CREATE OR REPLACE PACKAGE dynsql
IS
    invalid_table_name EXCEPTION;
    PRAGMA EXCEPTION_INIT (invalid_table_name, -903);
    invalid_identifier EXCEPTION;
    PRAGMA EXCEPTION_INIT (invalid_identifier, -904);
```

Теперь перехват этих ошибок в программе может производиться следующим образом:

```
WHEN dynsql.invalid identifier THEN ...
```

Аналогичный подход рекомендуется использовать при работе с кодами ошибок –20NNN, передаваемыми процедуре RAISE_APPLICATION_ERROR (см. далее в этой главе). Создайте пакет, в котором этим кодам будут присваиваться имена. Он может выглядеть примерно так:

```

PACKAGE errnums
IS
    en_too_young CONSTANT NUMBER := -20001;
    exc_too_young EXCEPTION;
    PRAGMA EXCEPTION_INIT (exc_too_young, -20001);
    en_sal_too_low CONSTANT NUMBER := -20002;
    exc_sal_too_low EXCEPTION;
    PRAGMA EXCEPTION_INIT (exc_sal_too_low , -20002);
END errnums;
```

При наличии такого пакета можно использовать код следующего вида, не указывая номер ошибки в коде:

```

PROCEDURE validate_emp (birthdate_in IN DATE)
IS
    min_years CONSTANT PLS_INTEGER := 18;
BEGIN
    IF ADD_MONTHS (SYSDATE, min_years * 12 * -1) < birthdate_in
    THEN
        RAISE_APPLICATION_ERROR
            (errnums.en_too_young,
             'Возраст работника должен быть не менее ' || min_years || ' лет.');
```

END IF;

END;

Именованные системные исключения

В Oracle для относительно небольшого количества исключений определены стандартные имена, задаваемые директивой компилятора `EXCEPTION_INIT` во встроенных пакетах. Самые важные и часто применяемые из них определены в пакете `STANDARD`. Так как это один из двух используемых по умолчанию пакетов `PL/SQL`, на определенные в нем исключения можно ссылаться без префикса с именем пакета. Например, если потребуется инициировать в программе исключение `NO_DATA_FOUND`, это можно сделать любой из следующих команд:

```
WHEN NO_DATA_FOUND THEN
WHEN STANDARD.NO_DATA_FOUND THEN
```

Определения стандартных именованных исключений встречаются и в других встроенных пакетах — например, в пакете `DBMS_LOB`, предназначенном для работы с большими объектами. Пример одного такого определения из указанного пакета:

```
invalid_argval EXCEPTION;
PRAGMA EXCEPTION_INIT(invalid_argval, -21560);
```

Поскольку пакет `DBMS_LOB` не используется по умолчанию, перед ссылкой на это исключение необходимо указать имя пакета:

```
WHEN DBMS_LOB.invalid_argval THEN...
```

Многие исключения, определенные в пакете `STANDARD`, перечислены в табл. 6.1. Для каждого из них приводится номер ошибки Oracle, значение, возвращаемое при вызове `SQLCODE` (встроенная функция `SQLCODE`, которая возвращает текущий код ошибки — см. раздел «Встроенные функции ошибок»), и краткое описание. Значение, возвращаемое `SQLCODE`, совпадает с кодом ошибки Oracle, с одним исключением: определяемый стандартом ANSI код ошибки `NO_DATA_FOUND` равен 100.

Таблица 6.1. Стандартные исключения в PL/SQL

| Имя исключения/Ошибка Oracle/ SQLCODE | Описание |
|--|---|
| <code>CURSOR_ALREADY_OPEN</code> ORA-6511 SQLCODE = -6511 | Попытка открытия курсора, который был открыт ранее. Перед повторным открытием курсор необходимо сначала закрыть |
| <code>DUP_VAL_ON_INDEX</code> ORA-00001 SQLCODE = -1 | Команда <code>INSERT</code> или <code>UPDATE</code> пытается сохранить повторяющиеся значения в столбцах, объявленных с ограничением <code>UNIQUE</code> |
| <code>INVALID_CURSOR</code> ORA-01001 SQLCODE = -1001 | Ссылка на несуществующий курсор. Обычно ошибка встречается при попытке выборки данных из неоткрытого курсора или закрытия курсора до его открытия |
| <code>INVALID_NUMBER</code> ORA-01722 SQLCODE = -1722 | Выполняемая SQL-команда не может преобразовать символьную строку в число. Это исключение отличается от <code>VALUE_ERROR</code> тем, что оно инициируется только из SQL-команд |
| <code>LOGIN_DENIED</code> ORA-01017 SQLCODE = -1017 | Попытка программы подключиться к СУБД Oracle с неверным именем пользователя или паролем. Исключение обычно встречается при внедрении кода PL/SQL в язык 3GL |
| <code>NO_DATA_FOUND</code> ORA-01403 SQLCODE = +100 | Исключение инициируется в трех случаях: (1) при выполнении инструкции <code>SELECT INTO</code> (невяный курсор), которая не возвращает ни одной записи; (2) при ссылке на неинициализированную запись локальной таблицы PL/SQL; (3) при попытке выполнить операцию чтения после достижения конца файла при использовании пакета <code>UTL_FILE</code> |
| <code>NOT_LOGGED_ON</code> ORA-01012 SQLCODE = -1012 | Программа пытается обратиться к базе данных (обычно из инструкции DML) до подключения к СУБД Oracle |
| <code>PROGRAM_ERROR</code> ORA-06501 SQLCODE = -6501 | Внутренняя программная ошибка PL/SQL. В сообщении об ошибке обычно предлагается обратиться в службу поддержки Oracle |

| Имя исключения/Ошибка Oracle/ SQLCODE | Описание |
|--|---|
| STORAGE_ERROR ORA-06500 SQLCODE = -6500 | Программе PL/SQL не хватает памяти или память по какой-то причине повреждена |
| TIMEOUT_ON_RESOURCE ORA-00051 SQLCODE = -51 | Тайм-аут СУБД при ожидании ресурса |
| TOO_MANY_ROWS ORA-01422 SQLCODE = -1422 | Команда SELECT INTO возвращает несколько записей, хотя должна возвращать лишь одну (в таких случаях инструкция SELECT включается в явное определение курсора, а записи выбираются по одной) |
| TRANSACTION_BACKED_OUT ORA-00061 SQLCODE = -61 | Удаленная часть транзакции отменена либо при помощи явной инструкции ROLLBACK, либо в результате какого-то другого действия (например, неудачного выполнения команды SQL или DML в удаленной базе данных) |
| VALUE_ERROR ORA-06502 SQLCODE = -6502 | Ошибка связана с преобразованием, усечением или проверкой ограничений числовых или символьных данных. Это общее и очень распространенное исключение. Если подобная ошибка содержится в инструкции SQL или DML, то в блоке PL/SQL инициируется исключение INVALID_NUMBER |
| ZERO_DIVIDE ORA-01476 SQLCODE = -1476 | Попытка деления на ноль |

Рассмотрим пример использования этой таблицы исключений. Предположим, ваша программа инициирует необрабатываемое исключение для ошибки ORA-6511. Заглянув в таблицу, вы видите, что она связана с исключением CURSOR_ALREADY_OPEN. Найдите блок PL/SQL, в котором произошла ошибка, и добавьте в него обработчик исключения CURSOR_ALREADY_OPEN:

```
EXCEPTION
    WHEN CURSOR_ALREADY_OPEN
    THEN
        CLOSE my_cursor;
END;
```

Конечно, еще лучше было бы проанализировать весь программный код и заранее определить, какие из стандартных исключений в нем могут инициироваться. В таком случае вы сможете решить, какие исключения следует обрабатывать конкретно, какие следует включить в конструкцию WHEN OTHERS (см. далее), а какие оставить необработанными.

Область действия исключения

Областью действия исключения называется та часть программного кода, к которой оно относится, то есть блок, где данное исключение может быть инициировано. В следующей таблице указаны области действия исключений четырех разных типов.

| Тип исключения | Область действия |
|--|--|
| Именованное системное исключение | Исключение является глобальным, то есть не ограничивается каким-то конкретным блоком кода. Системные исключения могут инициироваться и обрабатываться в любом блоке |
| Именованное исключение, определяемое программистом | Исключение может инициироваться и обрабатываться только в исполнительном разделе и разделе исключений, входящих в состав блока, где объявлено данное исключение (или в состав любого из вложенных в него блоков). Если исключение определено в спецификации пакета, то его областью действия являются все те программы, владельцы которых обладают для этого пакета привилегией EXECUTE |

продолжение ➤

| Тип исключения | Область действия |
|--|---|
| Анонимное системное исключение | Исключение может обрабатываться в секции WHEN OTHERS любого раздела исключений PL/SQL. Если присвоить ему имя, то его область действия будет такой же, как у именованного исключения, определяемого программистом |
| Анонимное исключение, определяемое программистом | Исключение определяется в вызове процедуры RAISE_APPLICATION_ERROR, а затем передается в вызывающую программу |

Рассмотрим пример исключения `overdue_balance`, объявленного в процедуре `check_account` (таким образом, область его действия ограничивается указанной процедурой):

```
PROCEDURE check_account (company_id_in IN NUMBER)
```

```
IS
```

```
    overdue_balance EXCEPTION;
```

```
BEGIN
```

```
    ... исполняемые команды ...
```

```
    LOOP
```

```
        ...
```

```
        IF ... THEN
```

```
            RAISE overdue_balance;
```

```
        END IF;
```

```
    END LOOP;
```

```
EXCEPTION
```

```
    WHEN overdue_balance THEN ...
```

```
END;
```

С помощью команды `RAISE` исключение `overdue_balance` можно инициировать в процедуре `check_account`, но не в программе, которая ее вызывает. Например, для следующего анонимного блока компилятор выдает ошибку:

```
DECLARE
```

```
    company_id NUMBER := 100;
```

```
BEGIN
```

```
    check_account (100);
```

```
EXCEPTION
```

```
    WHEN overdue_balance /* В PL/SQL такая ссылка недопустима. */
```

```
    THEN ...
```

```
END;
```

```
PLS-00201: identifier "OVERDUE_BALANCE" must be declared
```

Для приведенного выше анонимного блока процедура `check_account` является «черным ящиком». Все объявленные в ней идентификаторы, в том числе идентификаторы исключения, не видны для внешнего программного кода.

Инициирование исключений

Исключение может быть инициировано приложением в трех случаях:

- Oracle инициирует исключение при обнаружении ошибки;
- приложение инициирует исключение командой `RAISE`;
- исключение инициируется встроенной процедурой `RAISE_APPLICATION_ERROR`.

Как Oracle инициирует исключения, вы уже знаете. Теперь давайте посмотрим, как это может сделать программист.

Команда RAISE

Чтобы программист имел возможность самостоятельно инициировать именованные исключения, в Oracle поддерживается команда `RAISE`. С ее помощью можно инициировать как собственные, так и системные исключения. Команда имеет три формы:


```
RAISE имя_исключения;
RAISE имя_пакета.имя_исключения;
RAISE;
```

Первая форма (без имени пакета) может инициировать исключения, определенные в текущем блоке (или в содержащем его блоке), а также системные исключения, объявленные в пакете STANDARD. Далее приводятся два примера, в первом из которых иницируется исключение, определенное программистом:

```
DECLARE
    invalid_id EXCEPTION; -- Все идентификаторы должны начинаться с буквы 'X'.
    id_value VARCHAR2(30);
BEGIN
    id_value := id_for ('SMITH');
    IF SUBSTR (id_value, 1, 1) != 'X'
    THEN
        RAISE invalid_id;
    END IF;
    ...
END;
```

При необходимости вы всегда можете инициировать системное исключение:

```
BEGIN
    IF total_sales = 0
    THEN
        RAISE ZERO_DIVIDE; -- Определено в пакете STANDARD
    ELSE
        RETURN (sales_percentage_calculation (my_sales, total_sales));
    END IF;
END;
```

Если исключение объявлено в пакете (но не в STANDARD) и иницируется извне, имя исключения необходимо уточнить именем пакета:

```
IF days_overdue (isbn_in, borrower_in) > 365
THEN
    RAISE overdue_pkg.book_is_lost;
END IF;
```

Третья форма RAISE не требует указывать имя исключения, но используется только в условии WHEN раздела исключений. Ее синтаксис предельно прост:

```
RAISE;
```

Используйте эту форму для повторного инициирования (передачи) перехваченного исключения:

```
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        -- Используем общий пакет для сохранений всей контекстной
        -- информации: код ошибки, имя программы и т. д.
        errlog.putline (company_id_in);
        -- А теперь исключение NO_DATA_FOUND передается
        -- в родительский блок без обработки.
        RAISE;
```

Эта возможность особенно полезна в тех случаях, когда информацию об ошибке нужно записать в журнал, а сам процесс обработки возложить на родительский блок. Таким образом выполнение родительских блоков завершается без потери информации об ошибке.

Процедура RAISE_APPLICATION_ERROR

Для инициирования исключений, специфических для приложения, Oracle предоставляет процедуру RAISE_APPLICATION_ERROR (определенную в используемом по умолчанию пакете

DBMS_STANDARD). Ее преимущество перед командой RAISE (которая тоже может инициировать специфические для приложения явно объявленные исключения) заключается в том, что она позволяет связать с исключением сообщение об ошибке.

При вызове этой процедуры выполнение текущего блока PL/SQL прекращается, а любые изменения аргументов OUT и IN OUT (если таковые имеются) отменяются. Изменения, внесенные в глобальные структуры данных (с помощью команды INSERT, UPDATE, MERGE или DELETE), такие как переменные пакетов и объекты баз данных, не отменяются. Для отката DML-команд необходимо явно указать в разделе обработки исключений команду ROLLBACK.

Заголовок этой процедуры (определяемый в пакете DBMS_STANDARD) выглядит так:

```
PROCEDURE RAISE_APPLICATION_ERROR (
    num binary_integer,
    msg varchar2,
    keeperrorstack boolean default FALSE);
```

Здесь num — номер ошибки из диапазона от −20 999 до −20 000 (только представьте: все остальные отрицательные числа Oracle резервирует для собственных исключений!); msg — сообщение об ошибке, длина которого не должна превышать 2048 символов (символы, выходящие за эту границу, игнорируются); аргумент keeperrorstack указывает, хотите ли вы добавить ошибку к уже имеющимся в стеке (TRUE), или заменить существующую ошибку (значение по умолчанию — FALSE).



Oracle выделяет диапазон номеров от −20 999 до −20 000 для пользовательских ошибок, но учтите, что в некоторых встроенных пакетах, в том числе в DBMS_OUTPUT и DBMS_DESCRIBE, номера от −20 005 до −20 000 все равно присваиваются системным ошибкам. За дополнительной информацией обращайтесь к документации пакетов.

Рассмотрим пример полезного применения этой встроенной процедуры. Допустим, мы хотим, чтобы сообщения об ошибках выдавались пользователям на разных языках. Создадим для них таблицу error_table и определим в ней язык каждого сообщения значением столбца string_language. Затем создается процедура, которая генерирует заданную ошибку, загружая соответствующее сообщение из таблицы с учетом языка текущего сеанса:

```
/* Файл в Сети: raise_by_language.sp */
PROCEDURE raise_by_language (code_in IN PLS_INTEGER)
IS
    l_message error_table.error_string%TYPE;
BEGIN
    SELECT error_string
    INTO l_message
    FROM error_table
    WHERE error_number = code_in
    AND string_language = USERENV ('LANG');

    RAISE_APPLICATION_ERROR (code_in, l_message);
END;
```

Обработка исключений

Как только в программе возникает исключение, нормальное выполнение блока PL/SQL останавливается, и управление передается в раздел исключений. Затем исключение либо обрабатывается обработчиком исключений в текущем блоке PL/SQL, либо передается в родительский блок.

Чтобы обработать или перехватить исключение, нужно написать для него обработчик. Обработчики исключений располагаются после всех исполняемых команд блока, но перед завершающим ключевым словом **END**. Начало раздела исключений отмечает ключевое слово **EXCEPTION**:

```
DECLARE
    ... объявления ...
BEGIN
    ... исполняемые команды ...
[ EXCEPTION
    ... обработчики исключений ... ]
END;
```

Синтаксис обработчика исключений может быть таким:

```
WHEN имя_исключения [ OR имя_исключения ... ]
THEN
    исполняемые команды
```

или таким:

```
WHEN OTHERS
THEN
    исполняемые команды
```

В одном разделе исключений может быть несколько их обработчиков. Структура обработчиков напоминает структуру условной команды **CASE**.

| Свойство | Описание |
|---|---|
| EXCEPTION WHEN NO_DATA_FOUND THEN исполняемые_команды1; | Если инициировано исключение NO_DATA_FOUND, выполнить первый набор команд |
| WHEN payment_overdue THEN исполняемые_команды2; | Если просрочена оплата, выполнить второй набор команд |
| WHEN OTHERS THEN исполняемые_команды3; END; | Если инициировано иное исключение, выполнить третий набор команд |

Если имя, заданное в условии **WHEN**, совпадает с инициированным исключением, то это исключение обрабатывается соответствующим набором команд. Обратите внимание: исключения перехватываются по именам, а не по кодам ошибок. Но если инициированное исключение не имеет имени или его имя не соответствует ни одному из имен, указанных в условиях **WHEN**, тогда оно обрабатывается командами, заданными в секции **WHEN OTHERS** (если она имеется). Любая ошибка может быть перехвачена только одним обработчиком исключений. После выполнения команд обработчика управление сразу же передается из текущего блока в родительский или вызывающий блок.

Секция **WHEN OTHERS** не является обязательной. Когда она отсутствует, все необработанные исключения немедленно передаются в родительский блок, если таковой имеется. Секция **WHEN OTHERS** должна быть последним обработчиком исключений в блоке. Если разместить после нее еще одну секцию **WHEN**, компилятор выдаст сообщение об ошибке.

Встроенные функции ошибок

Прежде чем переходить к изучению тонкостей обработки ошибок, мы сначала вкратце познакомимся со встроенными функциями Oracle, предназначенными для идентификации, анализа и реагирования на ошибки, возникающие в приложениях PL/SQL.

○ SQLCODE

Функция **SQLCODE** возвращает код ошибки последнего исключения, инициированного в блоке. При отсутствии ошибок **SQLCODE** возвращает 0. Кроме того, **SQLCODE** возвращает 0 при вызове за пределами обработчика исключений.

База данных Oracle поддерживает стек значений `SQLCODE`. Допустим, к примеру, что функция `FUNC` инициирует исключение `VALUE_ERROR` (–6502). В разделе исключений `FUNC` вызывается процедура `PROC`, которая инициирует исключение `DUP_VAL_ON_INDEX` (–1). В разделе исключений `PROC` функция `SQLCODE` возвращает значение –1. Но когда управление передается в раздел исключений `FUNC`, `SQLCODE` будет возвращать –6502. Это поведение продемонстрировано в файле `sqlcode_test.sql` (доступен на сайте книги).

○ `SQLERRM`

Функция `SQLERRM` возвращает сообщение об ошибке для заданного кода ошибки. Если вызвать `SQLERRM` без указания кода ошибки, функция вернет сообщение, связанное со значением, возвращаемым `SQLCODE`. Например, если `SQLCODE` возвращает 0, функция `SQLERRM` вернет следующую строку:

```
ORA-0000: normal, successful completion
```

Если же `SQLCODE` возвращает 1 (обобщенный код ошибки для исключения, определяемого пользователем), `SQLERRM` вернет строку:

```
User-Defined Exception
```

Пример вызова `SQLERRM` для получения сообщения об ошибке для конкретного кода:

```
1 BEGIN
2     DBMS_OUTPUT.put_line (SQLERRM (-1403));
3* END;
SQL> /
ORA-01403: no data found
```

Максимальная длина строки, возвращаемой `SQLERRM`, составляет 512 байт (в некоторых ранних версиях Oracle — 255 байт). Из-за этого ограничения Oracle Corporation рекомендует вызывать функцию `DBMS_UTILITY.FORMAT_ERROR_STACK`, чтобы гарантировать вывод полной строки (эта встроенная функция не усекает текст до 2000 байт).

Примеры использования `SQLERRM` для проверки кодов ошибок представлены в файлах `oracle_error_info.pkg` и `oracle_error_info.tst` на сайте книги.

○ `DBMS_UTILITY.FORMAT_ERROR_STACK`

Эта встроенная функция, как и `SQLERRM`, возвращает сообщение, связанное с текущей ошибкой (то есть значение, возвращаемое `SQLCODE`). Ее отличия от `SQLERRM`:

- Она возвращает до 1899 символов сообщения, что позволяет избежать проблем с усечением.
- Этой функции не может передаваться код ошибки; соответственно, она не может использоваться для получения сообщения, соответствующего произвольному коду.

Как правило, эта функция вызывается в логике обработчика исключения для получения полного сообщения об ошибке.

Хотя в имя функции входит слово «stack», она не возвращает информацию о стеке ошибок, приведшем к строке, в которой изначально была инициирована ошибка. Эту задачу решает `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`.

○ `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`

Эта функция, появившаяся в Oracle10g, возвращает отформатированную строку с содержимым стека программ и номеров строк. Ее выходные данные позволяют отследить строку, в которой изначально была инициирована ошибка.

Тем самым заполняется весьма существенный пробел в функциональности PL/SQL. В Oracle9i и предшествующих версиях после обработки исключения в блоке PL/SQL было невозможно определить строку, в которой произошла ошибка (возможно, самая важная информация для разработчика). Если программист хотел получить эту информацию, он должен был разрешить прохождение необработанного исключения,

чтобы полная трассировочная информация ошибки была выведена на экран. Ситуация более подробно описана в следующем разделе.

○ DBMS_UTILITY.FORMAT_CALL_STACK

Функция возвращает отформатированную строку со стеком вызовов в приложении PL/SQL. Практическая полезность функции не ограничивается обработкой ошибок; она также пригодится для трассировки выполнения вашего кода. Функция более подробно описана в главе 20.



В Oracle Database 12c появился пакет UTL_CALL_STACK, который также предоставляет доступ к стеку вызовов, стеку ошибок и информации обратной трассировки. Этот пакет рассматривается в главе 20.

Подробнее о DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

Функцию DBMS_UTILITY.FORMAT_ERROR_BACKTRACE следует вызывать в обработчике исключения. Она выводит содержимое стека выполнения в точке инициирования исключения. Таким образом, вызов DBMS_UTILITY.FORMAT_ERROR_BACKTRACE в разделе исключений на верхнем уровне стека позволит узнать, где именно в стеке вызовов произошла ошибка. Рассмотрим следующий сценарий: мы определяем процедуру `proc3`, которая вызывает процедуру `proc2`, а последняя, в свою очередь, вызывает `proc1`. Процедура `proc1` иницирует исключение:

```
CREATE OR REPLACE PROCEDURE proc1 IS
BEGIN
    DBMS_OUTPUT.put_line ('выполнение proc1');
    RAISE NO_DATA_FOUND;
END;
/

CREATE OR REPLACE PROCEDURE proc2 IS
    l_str VARCHAR2 (30) := 'вызов proc1';
BEGIN
    DBMS_OUTPUT.put_line (l_str);
    proc1;
END;
/

CREATE OR REPLACE PROCEDURE proc3 IS
BEGIN
    DBMS_OUTPUT.put_line ('вызов proc2');
    proc2;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line ('Стек ошибок верхнего уровня:');
        DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace);
END;
/
```

Единственной программой с обработчиком ошибок является внешняя процедура `proc3`. Вызов функции трассировки включен в обработчик `WHEN OTHERS` процедуры `proc3`. При выполнении этой процедуры будет получен следующий результат:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
2     DBMS_OUTPUT.put_line ('Proc3 -> Proc2 -> Proc1 backtrace');
3     proc3;
4 END;
```

```

5 /
Proc3 -> Proc2 -> Proc1 backtrace
вызов proc2
вызов proc1
выполнение proc1
Error stack at top level:
ORA-06512: at "SCOTT.PROC1", line 4
ORA-06512: at "SCOTT.PROC2", line 5
ORA-06512: at "SCOTT.PROC3", line 4

```

Как видите, функция трассировки выводит в начале стека номер строки `proc1`, в которой произошла исходная ошибка.

Часто исключение происходит где-то в глубине стека вызовов. Если вы хотите, чтобы оно было передано во внешний блок PL/SQL, вероятно, вам придется заново инициировать его в каждом обработчике стека блоков. Функция `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE` выдает трассировку исполнения вплоть до последней команды `RAISE` в сеансе пользователя. Учтите, что вызов `RAISE` для конкретного исключения или повторное инициирование текущего исключения приводит к инициализации стека, выдаваемого `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`. Таким образом, если вы хотите использовать эту функцию, возможны два пути:

- Вызовите функцию в разделе исключений блока, в котором была инициирована ошибка. Это позволит вам получить (и сохранить в журнале) номер ошибки, даже если исключение было заново инициировано в дальнейшей позиции стека.
- Обойдите обработчики исключений в промежуточных программах вашего стека и вызовите функцию в разделе исключений внешней программы в стеке.

Только номер строки, пожалуйста

В реальном приложении трассировка ошибок может быть очень длинной. Как правило, специалиста, занимающегося отладкой или поддержкой, не интересует *весь* стек — достаточно только последнего элемента. Возможно, разработчику приложения стоит вывести эту важную информацию, чтобы пользователь мог немедленно и точно описать суть проблемы группе поддержки.

В такой ситуации необходимо разобрать строку с данными трассировки и извлечь из нее последний элемент. Я написал для этого специальную программу и оформил ее в пакет `BT`; вы можете загрузить ее на сайте книги. В этом пакете реализован простой, понятный интерфейс:

```

/* Файл в Сети: bt.pkg */
PACKAGE bt
IS
    TYPE error_rt IS RECORD (
        program_owner all_objects.owner%TYPE
        , program_name all_objects.object_name%TYPE
        , line_number PLS_INTEGER
    );

    FUNCTION info (backtrace_in IN VARCHAR2)
        RETURN error_rt;

    PROCEDURE show_info (backtrace_in IN VARCHAR2);
END bt;

```

Тип записи `error_rt` содержит отдельное поле для каждого возвращаемого элемента трассировки (владелец программного модуля, имя программного модуля и номер строки в программе). Затем вместо того, чтобы вызывать функцию трассировки в каждом разделе исключения и разбирать ее результаты, я вызываю функцию `bt.info` и вывожу конкретную информацию об ошибке.

Полезные применения SQLERRM

Вы можете использовать `DBMS_UTILITY.FORMAT_ERROR_STACK` вместо `SQLERRM`, но это не означает, что функция `SQLERRM` совершенно неактуальна. В частности, она поможет вам получить ответ на следующие вопросы:

- Является ли заданное число действительным кодом ошибки Oracle?
- Какое сообщение соответствует коду ошибки?

Как упоминалось ранее в этой главе, функция `SQLERRM` возвращает сообщение об ошибке для заданного кода. Но если передать `SQLERRM` недействительный код, исключение не инициируется. Вместо этого возвращается строка в одном из двух форматов:

- Если число отрицательно:
ORA-NNNNN: Message NNNNN not found; product=RDBMS; facility=ORA
- Если число положительно или меньше -65535:
-N: non-ORACLE exception

Этим обстоятельством можно воспользоваться для построения функций, возвращающих точную информацию о том коде, с которым вы работаете в настоящее время. Ниже приведена спецификация пакета с этими программами:

```
/* Файл в Сети: oracle_error_info.pkg */
PACKAGE oracle_error_info
IS
    FUNCTION is_app_error (code_in IN INTEGER)
        RETURN BOOLEAN;
    FUNCTION is_valid_oracle_error (
        code_in            IN    INTEGER
        , app_errors_ok_in IN    BOOLEAN DEFAULT TRUE
        , user_error_ok_in IN    BOOLEAN DEFAULT TRUE
    )
        RETURN BOOLEAN;
    PROCEDURE validate_oracle_error (
        code_in            IN    INTEGER
        , message_out      OUT    VARCHAR2
        , is_valid_out      OUT    BOOLEAN
        , app_errors_ok_in IN    BOOLEAN DEFAULT TRUE
        , user_error_ok_in IN    BOOLEAN DEFAULT TRUE
    );
END oracle_error_info;
```

Полная реализация приведена на сайте книги.

Объединение нескольких исключений в одном обработчике

В одном условии `WHEN` можно оператором `OR` объединить несколько исключений — подобно тому, как этим оператором объединяются логические выражения:

```
WHEN invalid_company_id OR negative_balance
THEN
```

В одном обработчике также можно комбинировать имена пользовательских и системных исключений:

```
WHEN balance_too_low OR ZERO_DIVIDE OR DBMS_LDAP.INVALID_SESSION
THEN
```

Впрочем, применять оператор `AND` в такой комбинации нельзя, потому что в любой момент времени может быть инициировано только одно исключение.

Необработанные исключения

Исключение, инициированное в программе, но не обработанное в соответствующем разделе текущего или родительского блока PL/SQL, называется *необработанным*. PL/SQL возвращает сообщение об ошибке, вызвавшей необработанное исключение, в ту среду, где была запущена данная программа. Эта среда (ею может быть SQL*Plus, Oracle Forms, программа на языке Java и т. д.) действует по ситуации. В частности, SQL*Plus осуществляет откат всех DML-инструкций, выполненных в родительском блоке.

Одним из важнейших моментов, связанных с проектированием архитектуры приложения, является вопрос о том, разрешается ли в нем использовать необработанные исключения. Такие исключения разными средами обрабатываются по-разному, и не всегда это делается корректно. Если ваша программа PL/SQL вызывается не из PL/SQL-среды, в ее «самом внешнем» блоке можно запрограммировать следующие действия:

- перехват всех исключений, которые могли быть переданы до текущей точки;
- запись информации об ошибке в журнал, с тем чтобы впоследствии ее мог проанализировать разработчик;
- возврат кода состояния, описания и другой информации, необходимой управляющей среде для выбора оптимального варианта действий.

Передача необработанного исключения

Блок, в котором может быть инициировано исключение, определяется правилами области действия исключений. В программе инициированное исключение распространяется в соответствии с определенными правилами.

Сначала PL/SQL ищет обработчик исключения в текущем блоке (анонимном блоке, процедуре или функции). Если такового нет, исключение передается в родительский блок. Затем PL/SQL пытается обработать исключение, инициировав его еще раз в родительском блоке. И так происходит в каждом внешнем по отношению к другому блоку до тех пор, пока все они не будут исчерпаны (рис. 6.2). После этого PL/SQL возвращает необработанное исключение в среду приложения, выполнившего «самый внешний» блок PL/SQL. И только теперь исключение может прервать выполнение основной программы.

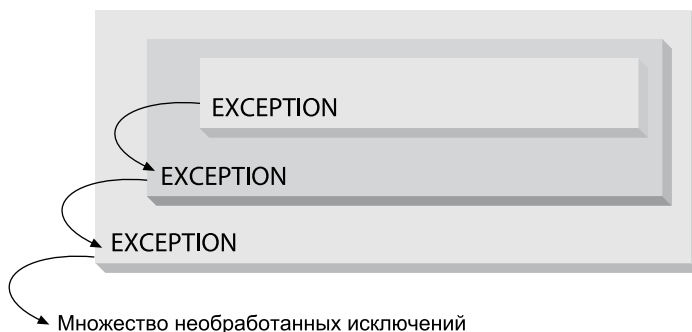


Рис. 6.2. Передача исключений во вложенных блоках

Потеря информации об исключении

Структура процесса обработки локальных, определяемых программистом исключений в PL/SQL такова, что можно легко потерять информацию об исключении (то есть о том, какая именно произошла ошибка). Пример:


```

BEGIN
  <<local_block>>
  DECLARE
    case_is_not_made EXCEPTION;
  BEGIN
    ...
  END local_block;

```

Допустим, мы забыли включить в этот блок раздел исключений. Область действия исключения `case_is_not_made` ограничена блоком `local_block`. Если исключение не обрабатывается в данном блоке, оно передается в родительский, где нет никакой информации о нем. Известно только то, что произошла ошибка, а *какая именно* — неизвестно. Ведь все пользовательские исключения имеют один и тот же номер ошибки 1 и одно и то же сообщение «User Defined Exception» — если только вы не воспользуетесь директивой `EXCEPTION_INIT`, чтобы связать с объявленным исключением другой номер, и не присвоите ему другое сообщение об ошибке при вызове `RAISE_APPLICATION_ERROR`.

Таким образом, локально объявленные (и инициированные) исключения всегда следует обрабатывать по имени.

Примеры передачи исключения

Рассмотрим несколько примеров передачи исключений через внешние блоки. На рис. 6.3 показано, как исключение `too_many_faults`, инициированное во внутреннем блоке, обрабатывается в следующем — внешнем — блоке. Внутренний блок содержит раздел исключений, так что PL/SQL сначала проверяет, обрабатывается ли в этом разделе инициированное исключение `too_many_faults`.

```

PROCEDURE list_my_faults IS
BEGIN

```

```

  DECLARE
    too_many_faults EXCEPTION;
  BEGIN
    ... executable statements before new block ...
    BEGIN
      SELECT SUM (faults) INTO num_faults FROM profile
      IF num_faults > 100
      THEN
        RAISE too_many_faults;
      END IF;
      EXCEPTION
        WHEN NO_DATA_FOUND THEN ;
      END;
    ... Исполняемые команды за Вложенным блоком 2 ...
  EXCEPTION
    WHEN too_many_faults THEN ... ;
  END;

```

Вложенный блок 1

Вложенный блок 2

```

END list_my_faults;

```

Рис. 6.3. Передача исключений во вложенных блоках

А поскольку оно не обрабатывается, PL/SQL закрывает этот блок и инициирует исключение `too_many_faults` во внешнем блоке, обозначенном на рисунке как вложенный блок 1. (Используемые команды, расположенные после вложенного блока 2, не выполняются.) Затем просматривается раздел исключений этого блока с целью поиска обработчика исключения `too_many_faults`, который обрабатывает его и передает управление процедуре `list_my_faults`.

Обратите внимание: если исключение `NO_DATA_FOUND` будет инициировано в «самом внутреннем» блоке, то оно будет обработано в разделе исключений этого же блока. Затем управление передается во вложенный блок 1 и будут выполнены исполняемые команды, расположенные после вложенного блока 2.

На рис. 6.4 представлен пример обработки в «самом внешнем» блоке исключения, инициированного во внутреннем блоке. В изображенной ситуации раздел исключений присутствует только во внешнем блоке, поэтому когда во вложенном блоке 2 инициируется исключение `too_many_faults`, PL/SQL прекращает выполнение этого блока и инициирует данное исключение в его родительском блоке, то есть вложенном блоке 1. Но поскольку и у него нет раздела исключений, управление передается «самому внешнему» блоку, процедуре `list_my_faults`. В этой процедуре имеется раздел исключений, поэтому PL/SQL проверяет его, находит обработчик исключения `too_many_faults`, выполняет имеющийся там код и передает управление программе, вызвавшей процедуру `list_my_faults`.

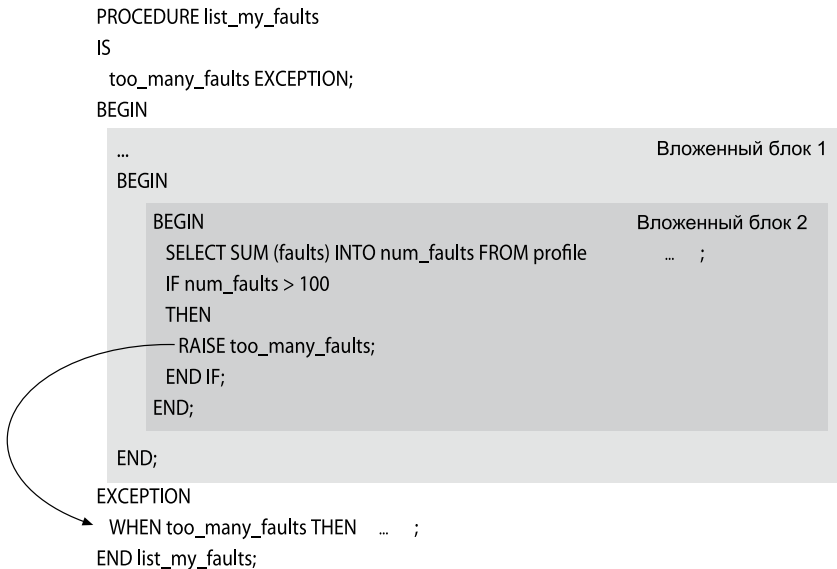


Рис. 6.4. Исключение, инициированное во вложенном блоке, обрабатывается в «самом внешнем» блоке

Продолжение выполнения после исключений

Когда в блоке PL/SQL инициируется исключение, нормальная последовательность выполнения программы прерывается, а управление передается в раздел исключений. Вернуться к исполняемому разделу блока после возникновения в нем исключения уже не удастся. Впрочем, в некоторых ситуациях требуется именно это — продолжить выполнение программы после обработки исключения.

Рассмотрим следующий сценарий: требуется написать процедуру, которая применяет серию операций DML к разным таблицам (удаление из одной таблицы, обновление другой, вставка в последнюю таблицу). На первый взгляд код мог бы выглядеть примерно так:

```
PROCEDURE change_data IS
BEGIN
    DELETE FROM employees WHERE ... ;
    UPDATE company SET ... ;
    INSERT INTO company_history SELECT * FROM company WHERE ... ;
END;
```

Безусловно, процедура содержит все необходимые команды DML. Однако одно из требований к программе заключается в том, что при последовательном выполнении этих команд они должны быть логически независимы друг от друга. Другими словами, даже если при выполнении `DELETE` произойдет сбой, программа должна выполнить `UPDATE` и `INSERT`.

В текущей версии `change_data` ничто не гарантирует, что программа хотя бы попытается выполнить все три операции DML. Если при выполнении `DELETE` произойдет исключение, например, то выполнение всей программы прервется, а управление будет передано в раздел исключений (если он имеется). Остальные команды SQL при этом выполняться не будут.

Как обеспечить обработку исключения без прерывания программы? Для этого `DELETE` следует поместить в собственный блок PL/SQL. Рассмотрим следующую версию программы `change_data`:

```
PROCEDURE change_data IS
BEGIN
    BEGIN
        DELETE FROM employees WHERE ... ;
    EXCEPTION
        WHEN OTHERS THEN log_error;
    END;

    BEGIN
        UPDATE company SET ... ;
    EXCEPTION
        WHEN OTHERS THEN log_error;
    END;

    BEGIN
        INSERT INTO company_history SELECT * FROM company WHERE ... ;
    EXCEPTION
        WHEN OTHERS THEN log_error;
    END;
END;
```

В новом варианте программы, если при выполнении `DELETE` произойдет исключение, управление немедленно передается в раздел исключений. Но поскольку команда `DELETE` теперь находится в собственном блоке, она может иметь собственный раздел исключений. Условие `WHEN OTHERS` этого раздела обрабатывает ошибку *без повторного инициирования этой или другой ошибки*, после чего управление возвращается за пределы блока `DELETE` внешней процедуре `change_data`. Так как «активное» исключение отсутствует, выполнение продолжается во внешнем блоке со следующей команды процедуры. Программа входит в новый анонимный блок для команды `UPDATE`. Если при выполнении `UPDATE` произойдет ошибка, она будет перехвачена условием `WHEN OTHERS` раздела исключений `UPDATE`. Далее управление будет возвращено процедуре `change_data`, которая перейдет к выполнению команды `INSERT` (также содержащейся в собственном блоке).

На рис. 6.5 показано, как выполняется этот процесс для двух последовательно выполняемых команд DELETE.

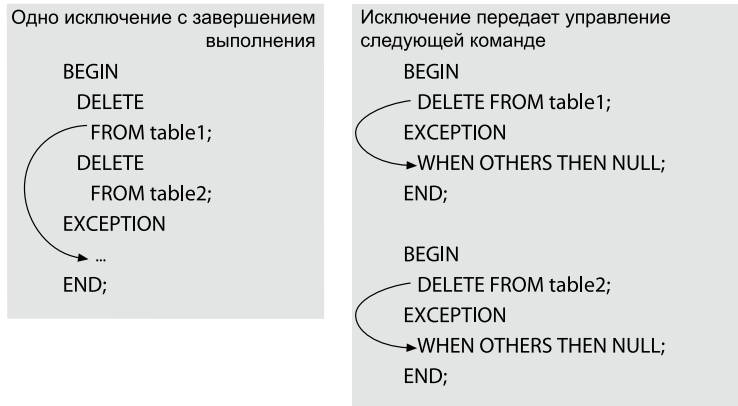


Рис. 6.5. Последовательное выполнение DELETE с разными областями действия

Подведем итог: исключение, инициированное в исполняемом разделе, всегда обрабатывается в текущем блоке (при наличии подходящего обработчика). Любую команду можно заключить в «виртуальный блок», заключив ее между ключевыми словами **BEGIN** и **END** с определением раздела **EXCEPTION**. Это позволяет ограничить область действия сбоев в программе посредством определения «буферных» анонимных блоков.

Эту стратегию можно развить с выделением изолируемого кода в отдельные процедуры и функции. Конечно, именованные блоки PL/SQL тоже могут иметь собственные разделы исключений и предоставлять ту же защиту от общих сбоев. Важнейшее преимущество процедур и функций заключается в том, что они скрывают все команды **BEGIN-EXCEPTION-END** от основной программы. Программа лучше читается, код проще сопровождать и повторно использовать в других контекстах.

Существуют и другие способы продолжить выполнение после исключения DML — например, можно использовать конструкцию **SAVE EXCEPTIONS** с **FORALL** и **LOG ERRORS** в сочетании с **DBMS_ERRORLOG**.

Написание раздела WHEN OTHERS

Условие **WHEN OTHERS** включается в раздел исключений для перехвата всех исключений, не обработанных предшествующими обработчиками. Так как конкретный тип исключения изначально неизвестен, в **WHEN OTHERS** очень часто используются встроенные функции для получения информации о возникшей ошибке (такие, как **SQLCODE** и **DBMS_UTILITY.FORMAT_ERROR_STACK**).

В сочетании с **WHEN OTHERS** функция **SQLCODE** представляет средства для обработки разных видов исключений без применения директивы **EXCEPTION_INIT**. В следующем примере перехватываются два исключения категории «родитель/потомок», -1 и -2292, и для каждой ситуации выполняется подходящее действие:

```

PROCEDURE add_company (
    id_in      IN company.ID%TYPE
    , name_in   IN company.name%TYPE
    , type_id_in IN company.type_id%TYPE
)
IS
```

```

BEGIN
  INSERT INTO company (ID, name, type_id)
    VALUES (id_in, name_in, type_id_in);
EXCEPTION
  WHEN OTHERS
  THEN
    /*
    || Анонимный блок в обработчике исключения позволяет объявить
    || локальные переменные для хранения информации о кодах ошибок.
    */
    DECLARE
      l_errcode PLS_INTEGER := SQLCODE;
    BEGIN
      CASE l_errcode
      WHEN -1 THEN
        -- Дублирующееся значение уникального индекса. Повторяется либо
        -- первичный ключ, либо имя. Сообщить о проблеме
        -- и инициировать исключение заново.
        DBMS_OUTPUT.put_line
          ( 'идентификатор или имя компании уже используется. ID = '
            || TO_CHAR (id_in)
            || ' name = '
            || name_in
          );
        RAISE;
      WHEN -2291 THEN
        -- Родительский ключ не найден. Сообщить о проблеме
        -- и инициировать исключение заново.
        DBMS_OUTPUT.put_line (
          'Недопустимый идентификатор типа компании: ' || TO_CHAR (type_id_in));
        RAISE;
      ELSE
        RAISE;
      END CASE;
    END; -- Конец анонимного блока.
END add_company;

```

Будьте осторожны при использовании `WHEN OTHERS` — этот раздел способен «поглощать» ошибки, скрывая их от внешних блоков и пользователя. А точнее, обращайте внимание на обработчики `WHEN OTHERS`, которые не инициируют текущее исключение заново и не заменяют его другим исключением. Если `WHEN OTHERS` не передает исключение наружу, внешние блоки вашей программы не узнают о возникшей ошибке.

В Oracle Database 11g появилось новое предупреждение, которое помогает выявлять программы, игнорирующие ошибки или поглощающие их:

```
PLW-06009: procedure "string" OTHERS handler does not end in RAISE or RAISE_
APPLICATION_ERROR
```

Пример использования этого предупреждения:

```

/* Файл в Сети: plw6009.sql */
SQL> ALTER SESSION SET plsql_warnings = 'enable:all'
2 /

SQL> CREATE OR REPLACE PROCEDURE plw6009_demo
2 AS
3 BEGIN
4   DBMS_OUTPUT.put_line ('I am here!');
5   RAISE NO_DATA_FOUND;
6 EXCEPTION
7   WHEN OTHERS
8   THEN
9     NULL;
10 END plw6009_demo;
11 /

```

продолжение ➤

SP2-0804: Procedure created with compilation warnings

SQL> SHOW ERRORS

Errors for PROCEDURE PLW6009_DEMO:

LINE/COL ERROR

```
-----
7/9      PLW-06009: procedure "PLW6009_DEMO" OTHERS handler does not end
         in RAISE or RAISE_APPLICATION_ERROR
```

Построение эффективной архитектуры управления ошибками

Механизм инициирования и обработки ошибок в PL/SQL отличается мощью и гибкостью, но он не лишен недостатков, которые могут создать проблемы для групп разработки, желающих реализовать надежную, последовательную, содержательную архитектуру управления ошибками. В частности, вы столкнетесь со следующими проблемами:

- **EXCEPTION** — особая разновидность структуры данных PL/SQL. Переменные, объявленные с типом **EXCEPTION**, можно только инициировать и обрабатывать. Исключение нельзя передать в аргументе программы, с ним нельзя связать дополнительные атрибуты.
- Повторное использование кода обработки исключений сильно затруднено. Из предыдущего пункта непосредственно следует другой факт: раз исключение нельзя передать в аргументе, разработчику приходится копировать код обработчика — конечно, такой способ написания кода никак не назовешь оптимальным.
- Не существует формализованного способа объявления исключений, которые могут инициироваться программой. Например, в Java эта информация становится частью спецификации программы. Как следствие, разработчику приходится обращаться к коду реализации и искать в нем информацию о потенциальных исключениях — или же надеяться на лучшее.
- Oracle не предоставляет средств организации и классификации исключений, относящихся к конкретному приложению, а просто резервирует (в основном) 1000 кодов в диапазоне от -20 999 до -20 000. Управлять этими значениями должен сам разработчик.

Давайте посмотрим, как преодолеть большинство из перечисленных трудностей.

Определение стратегии управления ошибками

Очень важно, чтобы еще до написания кода была выработана последовательная стратегия и архитектура обработки ошибок в приложении. Вот лишь некоторые вопросы, на которые необходимо ответить для этого:

- Как и когда сохранять информацию об ошибках для последующего просмотра и исправления? Куда выводить информацию — в файл, в таблицу базы данных? Выводить на экран?
- Как и где сообщать об ошибках пользователю? Какую информацию должен получать пользователь? Как «перевести» часто невразумительные сообщения об ошибках, выдаваемые базой данных, на язык, понятный пользователям?

С этими общими вопросами тесно связаны более конкретные проблемы:

- Следует ли включать раздел обработки исключений в каждый блок PL/SQL?
- Следует ли включать раздел обработки исключений только в блок верхнего уровня или внешние блоки?

○ Как организовать управление транзакциями при возникновении ошибок?

Сложность обработки исключений отчасти связана с тем, что на все эти вопросы не существует единственно правильного ответа. Все зависит (по крайней мере частично) от архитектуры приложения и режима его использования (например, пакетное выполнение или транзакции, управляемые пользователем). Но если вы сможете ответить на эти вопросы для своего приложения, я рекомендую «запрограммировать» стратегию и правила обработки ошибок в стандартном пакете (см. далее «Стандартизация обработки ошибок»).

Некоторые общие принципы, которые стоит принять во внимание:

- Когда в коде происходит ошибка, получите как можно больше информации о контексте ее возникновения. Избыток информации — лучше, чем ее нехватка. Далее исключение можно передавать во внешние блоки, собирая дополнительную информацию по мере продвижения.
- Избегайте применения обработчиков вида `WHEN ошибка THEN NULL`; (или еще хуже, `WHEN OTHERS THEN NULL`;). Возможно, для написания такого кода у вас имеются веские причины, но вы должны твердо понимать, что это именно то, что вам нужно, и документировать такое использование, чтобы о нем знали другие.
- Там, где это возможно, используйте механизмы обработки ошибок PL/SQL по умолчанию. Избегайте написания программ, возвращающих коды состояния управляющей среде или вызывающим блокам. Применять коды состояния следует только в одной ситуации: если управляющая среда не способна корректно обрабатывать ошибки Oracle (в таком случае стоит подумать о смене управляющей среды!).

Стандартизация обработки разных типов исключений

Исключение всегда свидетельствует о критической ситуации? Вовсе нет. Некоторые исключения (например, ORA-00600) сообщают о том, что в базе данных возникли очень серьезные низкоуровневые проблемы. Другие исключения, такие как `NO_DATA_FOUND`, встречаются так часто, что мы воспринимаем их не как ошибки, а как условную логическую конструкцию («Если строка не существует, то выполнить следующие действия...»). Нужно ли различать эти категории исключений?

Я считаю, что нужно, и Брин Луэллин (Bryn Llewellyn), руководитель разработки PL/SQL на момент написания книги, научил меня очень полезной системе классификации исключений.

- **Преднамеренные исключения.** Архитектура кода сознательно использует особенности работы исключения. Это означает, что разработчик должен предвидеть исключение и запрограммировать его обработку. Пример — `UTL_FILE.GET_LINE`.
- **Нежелательные исключения.** Происходит ошибка, но ее возможность была предусмотрена заранее. Возможно, исключение даже не свидетельствует о возникновении проблемы. Пример — команда `SELECT INTO`, инициирующая исключение `NO_DATA_FOUND`.
- **Непредвиденные исключения.** Серьезные ошибки, указывающие на возникновение проблемы в приложении. Пример — команда `SELECT INTO`, которая должна вернуть строку для заданного первичного ключа, но вместо этого инициирует исключение `TOO_MANY_ROWS`.

Давайте поближе познакомимся с примерами всех категорий, а затем поговорим о том, какую пользу вы можете извлечь из знания этих категорий.

Преднамеренные исключения

Разработчики PL/SQL могут использовать процедуру `UTL_FILE.GET_LINE` для чтения содержимого файла по строкам. Когда `GET_LINE` выходит за границу файла, инициируется исключение `NO_DATA_FOUND`. Так работает эта процедура. Итак, если я хочу прочитать все содержимое файла и сделать «что-то полезное», программа может выглядеть так:

```
PROCEDURE read_file_and_do_stuff (
    dir_in IN VARCHAR2, file_in IN VARCHAR2
)
IS
    l_file    UTL_FILE.file_type;
    l_line    VARCHAR2 (32767);
BEGIN
    l_file := UTL_FILE.fopen (dir_in, file_in, 'R', max_linesize => 32767);

    LOOP
        UTL_FILE.get_line (l_file, l_line);
        do_stuff;
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        UTL_FILE.fclose (l_file);
        more_stuff_here;
END;
```

У этого цикла есть одна особенность: он не содержит команды `EXIT`. Кроме того, в разделе исключений выполняется дополнительная логика приложения (`more_stuff_here`). Цикл можно переписать в следующем виде:

```
LOOP
    BEGIN
        UTL_FILE.get_line (l_file, l_line);
        do_stuff;
    EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
            EXIT;
    END;
END LOOP;

UTL_FILE.fclose (l_file);
more_stuff_here;
```

Теперь цикл содержит команду `EXIT`, но код стал более громоздким.

Подобные конструкции приходится использовать при работе с кодом, намеренно инициирующем исключения в своей архитектуре. Дополнительная информация о том, как следует поступать в подобных случаях, приводится в следующих разделах.

Нежелательные и непредвиденные исключения

Я рассматриваю эти две категории вместе, потому что приводимые примеры (`NO_DATA_FOUND` и `TOO_MANY_ROWS`) тесно связаны между собой. Предположим, я хочу написать функцию, возвращающую полное имя работника (в формате *фамилия запятая имя*) для заданного значения первичного ключа. Проще всего это сделать так:

```
FUNCTION fullname (
    employee_id_in IN employees.employee_id%TYPE
)
RETURN VARCHAR2
IS
    retval    VARCHAR2 (32767);
```



```
BEGIN
  SELECT last_name || ',' || first_name
    INTO retval
    FROM employees
   WHERE employee_id = employee_id_in;

  RETURN retval;
END fullname;
```

Если вызвать эту программу с кодом работника, отсутствующим в таблице, база данных инициирует исключение `NO_DATA_FOUND`. Если же вызвать ее с кодом работника, встречающимся в нескольких строках таблицы, будет инициировано исключение `TOO_MANY_ROWS`.

Один запрос, два разных исключения — нужно ли рассматривать их одинаково? Вероятно, нет. Описывают ли эти два исключения похожие группы проблем? Давайте посмотрим:

- **`NO_DATA_FOUND`** — совпадение не найдено. Исключение может указывать на наличие серьезной проблемы, но не обязательно. Возможно, в большинстве обращений к базе данных совпадение не будет обнаруживаться, и я буду вставлять в базу данные нового работника. В общем, исключение нежелательно, но в данном случае оно даже не указывает на возникновение ошибки.
- **`TOO_MANY_ROWS`** — в базе данных возникла серьезная проблема с ограничением первичного ключа. Трудно представить себе ситуацию, в которой это было бы нормально или просто «нежелательно». Нет, нужно прервать работу программы и привлечь внимание пользователя к совершенно непредвиденной, критической ошибке.

Как извлечь пользу из этой классификации

Надеюсь, вы согласитесь, что такая классификация полезна. Приступая к построению нового приложения, постарайтесь по возможности определиться со стандартным подходом, который будет применяться вами (и всеми остальными участниками группы) для каждого типа исключений. Затем для каждого исключения (которое необходимо обработать или хотя бы учитывать заранее при написании кода) решите, к какой категории относится, и примените уже согласованный подход. Все это поможет сделать ваш код более последовательным, и повысит эффективность вашей работы.

Приведу несколько рекомендаций для трех типов исключений.

- **Преднамеренные исключения.** Пишите код, учитывающий возможность возникновения таких исключений. Прежде всего постарайтесь избежать размещения логики приложения в разделе исключений. Раздел исключений должен содержать только код, относящийся к обработке ошибки: сохранение информации об ошибке в журнале, повторное инициирование исключения и т. д. Программисты не ожидают увидеть логику приложения в разделе исключений, поэтому им будет намного труднее разобраться в таком коде и обеспечить его сопровождение.
- **Нежелательные исключения.** Если в каких-то обстоятельствах пользователь кода, инициировавшего исключения, не будет интерпретировать ситуацию как ошибку, не передавайте исключения наружу без обработки. Вместо этого верните значение или флаг состояния, показывающий, что исключение было обработано. Далее пользователь программы может сам решить, должна ли программа завершиться с ошибкой. А еще лучше — почему бы не разрешить стороне, вызывающей вашу программу, решить, нужно ли инициировать исключение, и если не нужно — какое значение должно передаваться для обозначения возникшего исключения?
- **Непредвиденные исключения.** А теперь начинается самое неприятное. Все непредвиденные ошибки должны быть сохранены в журнале с максимумом возможной контекстной информации, которая поможет понять причины возникновения ошибки. Затем программа должна завершиться с необработанным исключением (обычно тем

же), инициированным из программы; для этого можно воспользоваться командой `RAISE`. Исключение заставит вызвавшую программу прервать работу и обработать ошибку.

Коды ошибок, связанные с конкретным приложением

Используя команду `RAISE_APPLICATION_ERROR` для инициирования ошибок, относящихся к конкретному приложению, вы несете полную ответственность за управление кодами ошибок и сообщениями. Это быстро становится хлопотным и непростым делом («Так, какой бы код мне выбрать? Пожалуй, `-20 774` — вроде бы такого еще не было?»).

Чтобы упростить управление кодами ошибок и предоставить последовательный интерфейс, через который разработчики смогут обрабатывать серверные ошибки, постройте таблицу со всеми используемыми кодами ошибок `-20 NNN`, сопутствующими именами исключений и сообщениями об ошибках.

Разработчик может просмотреть уже определенные ошибки на экране и выбрать ту из них, которая лучше всего подходит для конкретной ситуации. Пример такой таблицы (с кодом, генерирующим пакет с объявлениями всех «зарегистрированных» исключений) содержится в файле `msginfo.sql` на сайте книги.

Также можно попытаться полностью избегать диапазон `-20 NNN` для ошибок приложений. Почему бы не воспользоваться положительными числами? Из положительного целочисленного поддиапазона Oracle использует только 1 и 100. Теоретически *возможно*, что когда-нибудь Oracle будет использовать и другие положительные числа, но это весьма маловероятно. В распоряжении разработчиков остается великое множество кодов ошибок.

В частности, я пошел по этому пути при проектировании Quest Error Manager (QEM) — бесплатной программы управления ошибками. В Quest Error Manager вы можете определять свои ошибки в специальной таблице. Ошибка определяется именем и/или кодом. Коды ошибок могут быть положительными или отрицательными. Если код ошибки положителен, при инициировании исключения QEM использует команду `RAISE_APPLICATION_ERROR` для инициирования обобщенного исключения (обычно `-20 000`). Информация о текущем коде ошибки приложения встраивается в сообщение об ошибке, которое может быть расшифровано программой-получателем.

Упрощенная реализация этого подхода представлена в пакете обработки ошибок `errpkg`, описанном в следующем разделе.

Стандартизация обработки ошибок

Обязательным элементом любого профессионально написанного приложения является надежная и согласованная схема обработки ошибок. Согласованность в этом вопросе важна как для пользователя, так и для разработчика. Если при возникновении ошибки пользователю предоставляется понятная, хорошо структурированная информация, он сможет более подробно рассказать об ошибке службе поддержки и будет более уверенно чувствовать себя при работе с приложением. Если приложение всегда обрабатывает и протоколирует ошибки определенным образом, программистам, занимающимся его поддержкой и сопровождением, будет легче их найти и устранить.

Все кажется вполне очевидным, не так ли? К сожалению, на практике (и особенно в больших группах разработчиков) все происходит несколько иначе. Очень часто каждый разработчик идет своим путем, следуя личным принципам и приемам, сохраняя информацию в произвольно выбранном формате и т. д. Одним словом, без стандартизации отладка и сопровождение приложений оборачиваются сущим кошмаром. Рассмотрим типичный пример:

```

EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    v_msg := 'Нет компании с идентификатором ' || TO_CHAR (v_id);
    v_err := SQLCODE;
    v_prog := 'fixdebt';
    INSERT INTO errlog VALUES
      (v_err,v_msg,v_prog,SYSDATE,USER);
  WHEN OTHERS
  THEN
    v_err := SQLCODE;
    v_msg := SQLERRM;
    v_prog := 'fixdebt';
    INSERT INTO errlog VALUES
      (v_err,v_msg,v_prog,SYSDATE,USER);
    RAISE;

```

На первый взгляд код выглядит вполне разумно. Если компания с заданным идентификатором не найдена, мы получаем значение `SQLCODE`, задаем имя программы и сообщение и записываем строку с информацией об ошибке в таблицу ошибок. Выполнение родительского блока продолжается, поскольку ошибка не критична. Если происходит любая другая ошибка, получаем ее код и соответствующее сообщение, задаем имя программы и записываем строку с информацией об ошибке в таблицу ошибок, а затем передаем исключение в родительский блок, чтобы остановить его выполнение (поскольку неизвестно, насколько критична эта ошибка).

Что же здесь не так? Чтобы подробно объяснить суть проблемы, достаточно взглянуть на код. В нем жестко закодированы все действия по обработке ошибок. В результате (1) код получается слишком объемистым, (2) его придется полностью переписывать при изменении схемы обработки ошибок. Обратите внимание еще и на тот факт, что информация об ошибке записывается в таблицу базы данных. Это означает, что запись в журнале становится частью логической транзакции. И если потребуется выполнить откат транзакции, записи в журнале ошибок будут утеряны.

Существует несколько способов избежать потери информации: можно записывать данные в файл или использовать автономные транзакции для сохранения журнала вне основной транзакции. Но как бы то ни было, код в случае его изменения придется исправлять в сотнях разных мест.

А теперь посмотрите, как этот же раздел исключений оформляется при использовании стандартизированного пакета:

```

EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    errpkg.record_and_continue (
      SQLCODE, 'Нет компании с идентификатором ' || TO_CHAR (v_id));
  WHEN OTHERS
  THEN
    errpkg.record_and_stop;
END;

```

Такой пакет обработки ошибок скрывает все подробности реализации; вы просто решаете, какая из процедур-обработчиков должна использоваться в конкретном случае, просматривая спецификацию пакета. Если требуется сохранить информацию об ошибке и продолжить работу, вызывается программа `record_and_continue`. Если же нужно сохранить информацию об ошибке и прервать выполнение родительского блока, вызывается программа `record_and_stop`. Мы не знаем, как эти программы сохраняют информацию об ошибке, как они останавливают работу родительского блока, то есть передают исключение, но для нас это и не важно. Главное, что все происходит так, как определено стандартами приложения.

Это дает вам возможность уделить больше времени разработке более интересных элементов приложения и не заниматься административной рутинной.

На сайте книги имеется файл `errpkg.pkg` с прототипом стандартизированного пакета обработки ошибок. Правда, прежде чем использовать его в приложениях, вам необходимо будет завершить его реализацию; это поможет составить ясное представление о том, как конструируются подобные утилиты.

Вы также можете воспользоваться намного более мощным (и тоже бесплатным) средством обработки ошибок Quest Error Manager. Важнейшая концепция, заложенная в основу QEM, заключается в возможности перехвата и протоколирования *экземпляров* ошибок, не только ошибок Oracle. QEM состоит из пакета PL/SQL и четырех таблиц для хранения информации об ошибках, возникающих в приложениях.

Работа с «объектами» исключений

Реализация типа данных EXCEPTION в Oracle имеет свои ограничения, о которых было рассказано выше. Исключение состоит из идентификатора (имени), с которым связывается числовой код и сообщение. Исключение можно инициировать, его можно обработать... и все. Теперь представьте, как та же ситуация выглядит в Java: все ошибки являются производными от единого класса `Exception`. Этот класс можно расширить, дополняя его новыми характеристиками, которые вы хотите отслеживать (стек ошибок, контекстные данные и т. д.). Объект, созданный на основе класса `Exception`, ничем не отличается от любых других объектов Java. Разумеется, он может передаваться в аргументах методов.

PL/SQL не позволяет делать ничего подобного со своими исключениями. Впрочем, этот факт не мешает вам реализовать свой «объект» исключения. Для этого можно воспользоваться объектными типами Oracle или реляционной таблицей, содержащей информацию об ошибке. Независимо от выбранной реализации очень важно различать определение ошибки (код ошибки – 1403, имя «данные не найдены», причина — «неявный курсор не нашел ни одной записи») и ее конкретный экземпляр (я попытался найти компанию с указанным именем, ни одной строки не найдено). Иначе говоря, существует всего одно определение исключения `NO_DATA_FOUND`, которое может существовать во множестве экземпляров. Oracle не различает эти два представления ошибки, но для нас это безусловно необходимо.

Пример простой иерархии объектов исключений продемонстрирует этот момент. Начнем с базового объектного типа всех исключений:

```
/* Файл в Сети: exception.ot */
CREATE TYPE exception_t AS OBJECT (
  name VARCHAR2(100),
  code INTEGER,
  description VARCHAR2(4000),
  help_text VARCHAR2(4000),
  recommendation VARCHAR2(4000),
  error_stack CLOB,
  call_stack CLOB,
  created_on DATE,
  created_by VARCHAR2(100)
)
NOT FINAL;
/
```

Затем базовый тип исключения расширяется для ошибок динамического SQL посредством добавления атрибута `sql_string`. При обработке ошибок динамического SQL очень важно сохранить строку, создавшую проблемы, для анализа в будущем:

```
CREATE TYPE dynsql_exception_t UNDER exception_t (
    sql_string CLOB )
    NOT FINAL;
/
```

А вот другой подтип `exception_t`, на этот раз относящийся к конкретной сущности приложения — работнику. Исключение, инициируемое для ошибок, относящихся к работникам, будет включать идентификатор работника и внешний ключ нарушенного правила:

```
CREATE TYPE employee_exception_t UNDER exception_t (
    employee_id INTEGER,
    rule_id INTEGER );
/
```

Полная спецификация иерархии объектов ошибок включает методы супертипа исключения, предназначенные для вывода информации об ошибках или ее записи в репозиторий. Вы можете самостоятельно завершить иерархию, определенную в файле `exception.ot`. Если вы не хотите работать с объектными типами, попробуйте использовать подход, использованный мной в QEM: я определяю таблицу определений ошибок (`Q$ERROR`) и другую таблицу экземпляров ошибок (`Q$ERROR_INSTANCE`), которая содержит информацию о конкретных экземплярах ошибок. Все контекстные данные экземпляра ошибки сохраняются в таблице `Q$ERROR_CONTEXT`.

Пример кода, который мог бы быть написан для QEM API:

```
WHEN DUP_VAL_ON_INDEX
THEN
    q$error_manager.register_error (
        error_name_in => 'DUPLICATE-VALUE'
        ,err_instance_id_out => l_err_instance_id
    );
    q$error_manager.add_context (
        err_instance_id_in => l_err_instance_id
        ,name_in => 'TABLE_NAME', value_in => 'EMPLOYEES'
    );
    q$error_manager.add_context (
        err_instance_id_in => l_err_instance_id
        ,name_in => 'KEY_VALUE', value_in => l_employee_id
    );
    q$error_manager.raise_error_instance (
        err_instance_id_in => l_err_instance_id);
END;
```

Если ошибка повторяющегося значения была вызвана ограничением уникального имени, я получаю идентификатор экземпляра ошибки `DUPLICATE-VALUE`. (Да, все верно: я использую имена ошибок, полностью обходя все проблемы, связанные с номерами ошибок.) Затем я добавляю контекстную информацию экземпляра (имя таблицы и значение первичного ключа, вызвавшее проблему). В завершение инициируется экземпляр ошибки, в результате чего исключение передается в следующий наружный блок.

По аналогии с передачей данных из приложения в репозиторий ошибок через API, вы также можете получить информацию об ошибке при помощи процедуры `get_error_info`. Пример:

```
BEGIN
    run_my_application_code;
EXCEPTION
    WHEN OTHERS
    THEN
        DECLARE
            l_error    q$error_manager.error_info_rt;
        BEGIN
```

продолжение ➤

```

q$error_manager.get_error_info (l_error);
DBMS_OUTPUT.put_line ('');
DBMS_OUTPUT.put_line ('Error in DEPT_SAL Procedure:');
DBMS_OUTPUT.put_line ('Code = ' || l_error.code);
DBMS_OUTPUT.put_line ('Name = ' || l_error.NAME);
DBMS_OUTPUT.put_line ('Text = ' || l_error.text);
DBMS_OUTPUT.put_line ('Error Stack = ' || l_error.error_stack);
END;

```

END;

Это лишь два из многих способов преодоления ограничений типа EXCEPTION в PL/SQL. Мораль: ничто не заставляет вас мириться с ситуацией по умолчанию, при которой с экземпляром ошибки связывается только код и сообщение.

Создание стандартного шаблона для обобщенной обработки ошибок

Невозможность передачи исключений программе сильно усложняет совместное использование разделов обработки ошибок в разных блоках PL/SQL. Одну и ту же логику обработчика нередко приходится записывать снова и снова, особенно при работе с конкретными функциональными областями — скажем, файловым вводом/выводом с UTL_FILE. В таких ситуациях стоит выделить время на создание *шаблонов* обработчиков.

Давайте поближе познакомимся с UTL_FILE (см. далее в главе 22). До выхода Oracle9i Database Release 2 в спецификации пакета UTL_FILE определялся набор исключений. Однако компания Oracle не стала предоставлять коды этих исключений через директиву EXCEPTION_INIT. А без обработки исключений UTL_FILE по имени SQLCODE не сможет разобратся, что пошло не так. Вероятно, в такой ситуации для программ UTL_FILE можно создать шаблон, часть которого выглядит так:

```

/* Файл в Сети: utlfilexc.sql */
DECLARE
  l_file_id  UTL_FILE.file_type;
  PROCEDURE cleanup (file_in IN OUT UTL_FILE.file_type
                    ,err_in IN VARCHAR2 := NULL)
  IS
  BEGIN
    UTL_FILE.fclose (file_in);
    IF err_in IS NOT NULL
    THEN
      DBMS_OUTPUT.put_line ('Обнаружена ошибка UTL_FILE:');
      DBMS_OUTPUT.put_line (err_in);
    END IF;
  END cleanup;
BEGIN
  -- Здесь размещается тело программы.
  -- Перед выходом необходимо прибрать за собой ...
  cleanup (l_file_id);
EXCEPTION
  WHEN UTL_FILE.invalid_path
  THEN
    cleanup (l_file_id, 'invalid_path');
    RAISE;
  WHEN UTL_FILE.invalid_mode
  THEN
    cleanup (l_file_id, 'invalid_mode');
    RAISE;
END;

```

Основные элементы шаблона:

- Программа выполнения завершающих действий, пригодная для повторного использования; гарантирует, что текущий файл будет закрыт до потери дескриптора файла.
- Преобразование именованного исключения в строку, которую можно сохранить в журнале или вывести на экран, чтобы пользователь точно знал, какая ошибка была инициирована.

Рассмотрим еще один пример шаблона, который удобно использовать при работе с UTL_FILE. В Oracle9i Database Release 2 появилась программа `FREMOVE` для удаления файлов. Пакет `UTL_FILE` предоставляет исключение `DELETE_FAILED`, инициируемое тогда, когда `FREMOVE` не удастся удалить файл. После тестирования программы я обнаружил, что `FREMOVE` может инициировать несколько возможных исключений, в числе которых:

- `UTL_FILE.INVALID_OPERATION` — удаляемый файл не существует.
- `UTL_FILE.DELETE_FAILED` — у вас (или у процесса Oracle) недостаточно привилегий для удаления файла, или попытка завершилась неудачей по другой причине.



Начиная с Oracle9i Database Release 2, `UTL_FILE` назначает коды ошибок всем своим исключениям, но вы все равно должны проследить за тем, чтобы при возникновении ошибки файлы были закрыты, и организовать последовательную обработку ошибок.

Итак, при использовании `UTL_FILE.FREMOVE` следует включать раздел обработчика исключения, который различает эти две ошибки:

```
BEGIN
    UTL_FILE.fremove (dir, filename);
EXCEPTION
    WHEN UTL_FILE.delete_failed
    THEN
        DBMS_OUTPUT.put_line (
            'Ошибка при попытке удаления: ' || filename || ' в ' || dir);
        -- Выполнение соответствующих действий...
    WHEN UTL_FILE.invalid_operation
    THEN
        DBMS_OUTPUT.put_line (
            'Не удалось найти и удалить: ' || filename || ' в ' || dir);
        -- Выполнение соответствующих действий...
END;
```

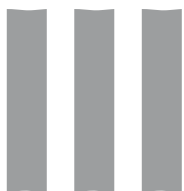
Файл `fileIO.pkg` на сайте книги содержит более полную реализацию такого шаблона для инкапсуляции `UTL_FILE.FREMOVE`.

Оптимальная организация обработки ошибок в PL/SQL

Без унифицированной качественной методологии обработки ошибок очень трудно написать приложение, которое было бы удобным в использовании и одновременно простым в отладке.

Архитектура обработки ошибок в Oracle PL/SQL предоставляет очень гибкие средства для определения, инициирования и обработки ошибок. Однако у нее имеются свои ограничения, вследствие чего встроенную функциональность обычно приходится дополнять таблицами и кодами ошибок, специфическими для конкретного приложения. Для решения проблемы обработки ошибок рекомендуется предпринять следующие действия:

1. Тщательно разберитесь в системе инициирования и обработки ошибок в PL/SQL. Далеко не во ее аспекты интуитивно понятны. Простейший пример: исключение, инициированное в секции объявлений, *не будет* обрабатываться секцией исключений текущего блока.
2. Выберите общую схему обработки ошибок в вашем приложении. Где и как будут обрабатываться ошибки? Какая информация об ошибке будет сохраняться и как это будет сделано? Как исключения будут передаваться в управляющую среду? Как будут обрабатываться намеренные и непредвиденные ошибки?
3. Постройте стандартную инфраструктуру, которая будет использоваться всеми разработчиками проекта. Инфраструктура должна включать таблицы, пакеты и, возможно, объектные типы, а также четко определенный процесс использования всех перечисленных элементов. Не останавливайтесь на ограничениях PL/SQL. Найдите обходные пути, расширяя модель обработки ошибок.
4. Создайте шаблоны, которые могут использоваться всеми участниками вашей группы. Всегда проще следовать готовому стандарту, чем самостоятельно писать код обработки ошибок.



Работа с данными в PL/SQL

Практически любая программа работает с данными, причем в основном эти данные являются *локальными*, то есть определяются в процедурах или функциях PL/SQL. В этой части книги описаны различные типы программных данных, которые используются в PL/SQL, — числа (включая новые типы данных, появившиеся в Oracle11g), строки, даты, коллекции типов данных XML и пользовательские типы данных. В главах 7–13 также рассматриваются различные встроенные функции Oracle, предназначенные для обработки и преобразования данных.

7

Работа с данными в программах

Практически любой написанный вами блок PL/SQL будет определять данные и выполнять с ними те или иные операции. Программные данные представляют собой структуры, которые существуют только в рамках сеанса PL/SQL (а физически — находятся в рамках глобальной области программ (PGA, Program Global Area)) и не хранятся в базе данных. К программным данным относятся:

- *Переменные и константы* — значения переменных могут изменяться во время выполнения программы, а значения констант статичны, то есть устанавливаются при объявлении и в дальнейшем не изменяются.
- *Скалярные и составные данные* — скалярные данные состоят из одного значения (например, числа или строки), а составные данные состоят из нескольких значений (как, например, запись, коллекция или экземпляр объектного типа).
- *Контейнеры* — могут содержать информацию, полученную из базы данных или иного источника.

Прежде чем работать с программными данными в коде PL/SQL, необходимо объявить структуры данных с указанием имен и типов данных.

В этой главе мы расскажем о том, как объявляются данные в программе, и опишем правила, которым нужно следовать при выборе имен для этих данных. Также будет приведена краткая сводка всех типов данных, поддерживаемых в PL/SQL, и рассмотрена концепция преобразования типов. Глава завершается рекомендациями о том, как лучше работать с программными данными. Конкретные типы данных подробно описаны в остальных главах этой части.

Присваивание имен

Чтобы использовать переменную или константу, ее необходимо сначала объявить, присвоив ей имя и определив тип. Ниже перечислены основные правила выбора имен в PL/SQL (они касаются также имен объектов базы данных, например таблиц или столбцов):

- длина имени не должна превышать 30 символов;
- имя должно начинаться с буквы и может состоять из букв, цифр, а также символов «\$», «#» и «_»;
- имена нечувствительны к регистру символов (если только они не заключены в двойные кавычки).

Согласно этим правилам допустимыми являются имена:

```
l_total_count
first_12_years
total_#_of_trees
salary_in_$
```

Следующие два имени допустимы, однако они считаются в PL/SQL идентичными из-за нечувствительности языка к регистру символов:

```
ExpertsExchange
ExpertSexChange
```

Следующие имена недопустимы по указанным причинам:

```
1st_account -- Начинается с цифры, а не буквы
favorite_ice_cream_flavors_that_dont_contain_nuts -- Слишком длинное имя
email_address@business_loc -- Имя содержит недопустимый символ @
```

У этих правил есть несколько исключений. Если имя в объявлении заключить в двойные кавычки, соблюдение этих правил не обязательно, кроме одного: длина имени не должна превышать 30 символов. Например, допустимыми являются следующие объявления:

```
DECLARE
    "truly_lower_case" INTEGER;
    " " DATE; -- Да, имя из пяти пробелов!
    "123_go!" VARCHAR2(10);
BEGIN
    "123_go!" := 'Steven';
END;
```

Когда эти имена указываются в исполняемом разделе, они всегда должны заключаться в двойные кавычки, иначе код не будет компилироваться.

Зачем нужны имена в двойных кавычках? В программах PL/SQL они не имеют особого смысла, но при создании объектов баз данных это позволяет сохранить в идентификаторах исходный регистр символов (например, если в программе создается таблица «docs», то она будет называться именно docs, а не DOCS). Однако в любых других случаях лучше избегать применения двойных кавычек в программах PL/SQL.

Другое исключение из правил имен относится к именам объектов Java, длина которых может достигать 4000 символов. За информацией об этих именах и о том, что они означают для разработчиков PL/SQL, обращайтесь к главе Java на сайте книги.

Выбирая имена для переменных и констант, придерживайтесь следующих рекомендаций.

- *Убедитесь в том, что имя соответствует назначению объекта и по нему можно быстро определить это назначение.* Попробуйте сформулировать, что представляет собой значение переменной; это поможет выбрать для нее более точное имя. Например, если переменная представляет «количество обращений по поводу еле теплого кофе», подходящим именем этой переменной может быть `total_calls_on_cold_coffee`, или `tot_cold_calls`, если вас раздражают имена переменных из пяти слов. Напротив, имена `totcoffee` или `t_#_calls_lwcoff` плохи — вряд ли читатель по имени переменной поймет, какие данные в ней хранятся.
- *Выработайте единые соглашения об именовании объектов и следуйте им.* Соглашения обычно предполагают использование префиксов и суффиксов, описывающих тип и назначение переменных. Например, имена всех локальных переменных могут начинаться с суффикса «`l_`», а имена глобальных переменных, определяемых в пакетах, — с префиксом «`g_`». Записям назначается суффикс «`_rt`» и т. д. Полный набор правил назначения имен можно загрузить с примерами кода моей книги «Oracle PL/SQL Best Practices».

Обзор типов данных PL/SQL

При объявлении переменной или константы необходимо задать ее тип данных (PL/SQL, за некоторыми исключениями, относится к языкам со статической типизацией — см. ниже врезку). В PL/SQL определен широкий набор скалярных и составных типов данных; кроме того, вы можете создавать пользовательские типы данных. Многие типы данных PL/SQL (например, `BOOLEAN` и `NATURAL`) не поддерживаются столбцами баз данных, но в коде PL/SQL эти типы весьма полезны.

Практически все заранее определенные типы данных определяются в пакете `PL/SQL STANDARD`. Например, определение типа данных `BOOLEAN` и двух числовых типов данных может выглядеть так:

```
create or replace package STANDARD is
```

```
    type BOOLEAN is (FALSE, TRUE);
```

```
    type NUMBER is NUMBER_BASE;  
    subtype INTEGER is NUMBER(38,);
```

PL/SQL поддерживает распространенный «джентльменский набор» типов данных, а также ряд других типов. В этом разделе приводится краткий обзор различных предопределенных типов данных; более подробные описания можно найти в главах 8–13, 15 и 26.

ЧТО ТАКОЕ «СТАТИЧЕСКАЯ ТИПИЗАЦИЯ»?

Статической (или сильной) типизацией называется проверка типов во время компиляции (а не на стадии выполнения программы). К числу языков программирования, использующих сильную типизацию, относятся PL/SQL, Ada, C и Pascal. Языки с динамической типизацией (такие, как JavaScript, Perl или Ruby) выполняют большинство проверок типов во время выполнения. Статическая типизация позволяет выявлять ошибки во время компиляции, что повышает надежность программ. Преимуществом статической типизации является и ускорение выполнения программ. Оптимизирующий компилятор, который знает точные типы данных, может подбирать более эффективные ассемблерные решения и генерировать высокооптимизированный машинный код. Динамическая типизация тоже обладает своими преимуществами: например, метаклассы и интроспекция проще реализуются на базе динамической типизации.

Символьные типы данных

PL/SQL поддерживает строки фиксированной и переменной длины, состоящие как из традиционных символов, так и из символов Юникода. К строкам первого типа относятся строки `CHAR` и `NCHAR`, а к строкам второго вида — `VARCHAR2` и `NVARCHAR2`. Объявление строки переменной длины, которая может содержать до 2000 символов, выглядит так:

```
DECLARE
```

```
    l_accident_description VARCHAR2(2000);
```

Правила работы с символьными данными, примеры их применения и встроенные функции, предназначенные для операций со строками в PL/SQL, описаны в главе 8.

Для очень длинных символьных строк в PL/SQL предусмотрены типы данных `CLOB` (Character Large Object) и `NCLOB` (NLS Character Large Object). По соображениям совместимости PL/SQL также поддерживает тип данных `LONG`. Эти типы данных позволяют

сохранять и обрабатывать очень большие объемы данных; так, в Oracle11g тип LOB способен хранить до 128 терабайт информации.



Использование типа LONG ограничивается множеством правил. Мы не рекомендуем применять его в Oracle8 и последующих версиях.

В главе 13 рассматриваются правила работы с большими объектами, приводится множество примеров их применения, а также описываются встроенные функции и пакет DBMS_LOB, предназначенный для обработки таких объектов средствами PL/SQL.

Числовые типы данных

В PL/SQL поддерживаются как вещественные, так и целочисленные типы данных. Тип NUMBER давно был основным типом для работы с числовыми данными; он может применяться для работы с целыми и вещественными данными как с фиксированной, так и с плавающей запятой. Пример типичного объявления NUMBER:

```
/* Файл в Сети: numbers.sql */
DECLARE
    salary NUMBER(9,2); -- фиксированная точка, семь цифр в целой части,
                        -- две в дробной части
    raise_factor NUMBER; -- дробное с плавающей запятой
    weeks_to_pay NUMBER(2); -- целое число
BEGIN
    salary := 1234567.89;
    raise_factor := 0.05;
    weeks_to_pay := 52;
END;
```

Из-за своей внутренней десятичной природы тип NUMBER особенно удобен при работе с денежными суммами. В отличие от двоичного представления, ему не присущи ошибки округления. Например, если сохранить в нем значение 0.95, позднее вы прочитаете именно эту величину, а не приближенную (скажем, 0.949999968).

До выхода Oracle10g тип NUMBER был единственным числовым типом данных PL/SQL, напрямую соответствовавшим типу столбцов базы данных. В этом нетрудно убедиться, изучив содержимое пакета STANDARD. Данная особенность стала одной из причин, по которым тип NUMBER так широко применялся в программах PL/SQL.

В Oracle10g появились два двоичных типа с плавающей запятой: BINARY_FLOAT и BINARY_DOUBLE. Как и NUMBER, эти двоичные типы данных поддерживаются и в PL/SQL, и в базах данных. Однако в отличие от NUMBER, эти типы хранят значение в двоичном виде, а следовательно, при работе с ними могут возникнуть погрешности округления. Типы BINARY_FLOAT и BINARY_DOUBLE поддерживают специальные значения NaN (Not a Number, «не является числом»), положительную и отрицательную бесконечность. В некоторых типах приложений эти типы обеспечивают огромный выигрыш в быстродействии, так как вычисления с этими двоичными типами по возможности выполняются на аппаратном уровне.

В Oracle11g появились еще две разновидности вещественных типов. Типы SIMPLE_FLOAT и SIMPLE_DOUBLE являются аналогами BINARY_FLOAT и BINARY_DOUBLE, но они не поддерживают NULL и не инициализируют исключение в случае переполнения.

PL/SQL поддерживает несколько числовых типов и подтипов, не имеющих прямого соответствия среди типов баз данных, но все равно полезных. Особого внимания

заслуживают `PLS_INTEGER` и `SIMPLE_INTEGER`. Операции с целочисленным типом `PLS_INTEGER` реализуются на аппаратном уровне. В частности, счетчики циклов `FOR` реализуются в формате `PLS_INTEGER`. Тип `SIMPLE_INTEGER`, появившийся в Oracle11g, обладает тем же диапазоном значений, что и `PLS_INTEGER`, но не поддерживает `NULL` и не иницирует исключение в случае переполнения. `SIMPLE_INTEGER`, как и `SIMPLE_FLOAT` с `SIMPLE_DOUBLE`, работает невероятно быстро — особенно с откомпилированным кодом.

Правила работы с числовыми данными, примеры их применения и встроенные функции, предназначенные для операций с числами в PL/SQL, описаны в главе 9.

Дата, время и интервалы

До появления версии Oracle9i представление дат в Oracle ограничивалось типом `DATE`, в котором хранится дата и время (с округлением до ближайшей секунды). В Oracle9i были введены два новых типа данных, `INTERVAL` и `TIMESTAMP`, значительно расширившие возможности разработчиков в отношении операций с датами и временем в PL/SQL, а также вычисления и хранения временных интервалов. Их использование продемонстрировано в функции, вычисляющей возраст человека в интервальном формате с точностью до месяца:

```
/* Файл в Сети: age.fnc */
FUNCTION age (dob_in IN DATE)
  RETURN INTERVAL YEAR TO MONTH
IS
BEGIN
  RETURN (SYSDATE - dob_in) YEAR TO MONTH;
END;
```

Правилам работы с датами, временем и интервалами посвящена глава 10. В ней также приведено много примеров и описаний соответствующих встроенных функций PL/SQL.

Логические данные

PL/SQL поддерживает тип данных `BOOLEAN`. Переменные этого типа могут принимать одно из трех значений (`TRUE`, `FALSE` или `NULL`).

Логические переменные помогают писать понятный, легко читаемый код даже в тех случаях, когда он содержит очень сложные логические выражения. Пример объявления логической переменной с присваиванием ей значения по умолчанию:

```
DECLARE
  l_eligible_for_discount BOOLEAN :=
    customer_in.balance > min_balance AND
    customer_in.pref_type = 'MOST FAVORED' AND
    customer_in.disc_eligibility;
```

Работа с логическими данными и примеры их использования описаны в главе 13.

Двоичные данные

Oracle поддерживает несколько разновидностей *двоичных данных* — неструктурированных данных, не интерпретируемых и не обрабатываемых Oracle. К их числу относятся типы `RAW`, `BLOB` и `BFILE`. Тип `BFILE` используется для хранения неструктурированных двоичных данных в файлах операционной системы вне базы данных. Тип `RAW` имеет переменную длину, а при операциях с ним Oracle не выполняет преобразование символов при передаче данных. В остальном он аналогичен символьному типу `VARCHAR2`.

Тип данных `LONG RAW` поддерживается для обеспечения обратной совместимости, но в PL/SQL поддержка данных `LONG RAW` ограничена. В базе данных Oracle столбец `LONG RAW`

занимает до 2 Гбайт, но PL/SQL позволяет работать только с первыми 32 760 байтами LONG RAW. Если, например, вы попытаетесь выполнить выборку в переменную PL/SQL из столбца LONG RAW, превышающего лимит в 32 760 байт, произойдет ошибка:

```
ORA-06502: PL/SQL: numeric or value error exception.
```

Для работы с данными LONG RAW, превышающими лимит PL/SQL, потребуется программа OCI; кстати, это веская причина для перевода старого кода с LONG RAW на данные BLOB, не имеющие такого ограничения.

В главе 13 рассматриваются правила работы с двоичными данными, приводятся многочисленные примеры, описываются встроенные функции и пакет DBMS_LOB, предназначенный для операций с данными типа BFILE и другими двоичными данными в PL/SQL.

Типы данных ROWID и UROWID

Oracle поддерживает два типа данных ROWID и UROWID, предназначенных для предоставления адреса строки в таблице. Тип ROWID представляет уникальный физический адрес строки в таблице, а тип UROWID — логическую позицию строки в индексной таблице (Index-Organized Table, IOT). Тип ROWID также является псевдостолбцом SQL, который может включаться в инструкции SQL.

Правила работы с типами данных ROWID и UROWID описаны в главе 13.

Тип данных REF CURSOR

Тип данных REF CURSOR позволяет объявлять курсорные переменные, которые могут использоваться со статическими и динамическими SQL-командами для улучшения гибкости программного кода. Тип REF CURSOR существует в двух формах: сильной и слабой. PL/SQL относится к категории языков со статической типизацией, а слабый тип REF CURSOR является одной из немногочисленных конструкций с динамической типизацией.

В следующем примере объявления сильной формы REF CURSOR курсорная переменная связывается с конкретной структурой записи с помощью атрибута %ROWTYPE:

```
DECLARE
    TYPE book_data_t IS REF CURSOR RETURN book%ROWTYPE;
    book_curs_var book_data_t;
```

Далее следуют два слабых объявления REF CURSOR, в которых переменная не связывается ни с какой конкретной структурой. Во втором объявлении (строка 4) представлен тип SYS_REFCURSOR — заранее определенный слабый тип REF CURSOR.

```
DECLARE
    TYPE book_data_t IS REF CURSOR;
    book_curs_var book_data_t;
    book_curs_var_b SYS_REFCURSOR;
```

Тип данных REF CURSOR и курсорные переменные рассматриваются в главе 15.

Типы данных для поддержки интернет-технологий

В Oracle9i появилась встроенная поддержка ряда технологий и типов данных, связанных с Интернетом, в частности XML (eXtensible Markup Language) и URI (Universal Resource Identifier). В Oracle имеются специализированные типы для работы с данными XML и URI, а также специальный класс идентификаторов URI (DBUri-REF), который используется для доступа к базе данных. Кроме того, в Oracle появился новый набор типов данных, обеспечивающий хранение внешних и внутренних URI и обращение к ним из базы данных.

Тип `XMLType` предназначен для запроса и сохранения данных в формате XML. Для работы с XML используются такие функции, как `SYS_XMLGEN` из пакета `DBMS_XMLGEN`. Поддержка XPath и встроенные команды языка SQL позволяют выполнять поиск данных в документах XML.

Типы данных для работы с URI, включая `URIType` и `HttpURIType`, входят в иерархию объектных типов и могут использоваться для хранения URI внешних веб-страниц и файлов, а также для обращения к информации в базе данных.

В главе 13 рассматриваются правила работы с типом данных `XMLType` и различными URI-типами, приводятся примеры их использования и описываются встроенные функции для работы с ними.

Типы данных «Any»

Большая часть программного кода предназначена для решения узкоспециализированных задач. Однако иногда нам приходится писать более универсальные программы. Именно для таких ситуаций предназначены типы данных `Any`.

Эта группа типов данных, появившаяся в Oracle9i, заметно отличается от любых других типов данных Oracle. Типы данных `Any` позволяют динамически инкапсулировать описания типов, экземпляры данных и наборы экземпляров данных любого другого типа SQL. С помощью этих объектных типов (и определенных для них методов) можно, к примеру, определить тип данных, хранимых во вложенной таблице, не обращаясь к объявлению типа этой таблицы!

В группу типов данных `Any` входят `AnyType`, `AnyData` и `AnyDataSet`.

Правила работы с типами данных `Any` и реальные примеры их использования представлены в главе 13.

Пользовательские типы данных

Из встроенных типов данных Oracle и пользовательских типов можно строить типы данных произвольной сложности, которые с большой точностью отражают структуру и поведение данных в конкретных системах.

Эта возможность подробно рассматривается в главе 26. Там же рассказано о том, как использовать добавленную в Oracle9i поддержку наследования обычных типов данных.

Объявление данных в программе

Прежде чем использовать переменную или константу в программе, ее почти всегда необходимо объявить. Все объявления должны размещаться в разделе объявлений программы PL/SQL. (Подробнее о структуре блока PL/SQL и его раздела объявлений рассказано в главе 3.)

В PL/SQL объявления могут относиться к переменным, константам, `TYPE` (например, коллекциям или записям) и исключениям. В этой главе рассматриваются объявления переменных и констант. (Команды `TYPE` для записей рассматриваются в главе 11, а для коллекций — в главе 12; объявления исключений рассматриваются в главе 6.)

Объявление переменной

Когда вы объявляете переменную, PL/SQL выделяет память для хранения ее значения и присваивает выделенной области памяти имя, по которому это значение можно

извлекать и изменять. В объявлении также задается тип данных переменной; он используется для проверки присваиваемых ей значений. Базовый синтаксис объявления переменной или константы:

```
имя тип_данных [NOT NULL] [ := | DEFAULT значение_по_умолчанию];
```

Здесь *имя* — имя переменной или константы, *тип_данных* — тип или подтип данных, определяющий, какие значения могут присваиваться переменной. При желании можно включить в объявление выражение `NOT NULL`; если такой переменной не присвоено значение, то база данных инициирует исключение. Секция *значение_по_умолчанию* инициализирует переменную начальным значением; ее присутствие обязательно только при объявлении констант. Если переменная объявляется с условием `NOT NULL`, то при объявлении ей должно быть присвоено начальное значение.

Примеры объявления переменных разных типов:

```
DECLARE
-- Простое объявление числовой переменной
l_total_count NUMBER;

-- Число, округляемое до двух разрядов в дробной части:
l_dollar_amount NUMBER (10,2);

-- Дата/время, инициализируемая текущим значением системных часов
-- сервера базы данных. Не может принимать значение NULL
l_right_now DATE NOT NULL DEFAULT SYSDATE;

-- Задание значения по умолчанию с помощью оператора присваивания
l_favorite_flavor VARCHAR2(100) := 'Вы любите кофе?';

-- Ассоциативный массив объявляется за два этапа.
-- Сначала тип таблицы:
TYPE list_of_books_t IS TABLE OF book%ROWTYPE INDEX BY BINARY_INTEGER;

-- А затем конкретный список, с которым мы будем работать в блоке:
oreilly_oracle_books list_of_books_t;
```

Конструкция `DEFAULT` (см. `l_right_now` в приведенном примере) и конструкция с оператором присваивания (`l_favorite_flavor` в приведенном примере) эквивалентны и взаимозаменяемы. Какой из двух вариантов использовать? Мы предпочитаем для констант использовать оператор присваивания (`:=`), а для инициализации переменных — ключевое слово `DEFAULT`. При объявлении константы задается не значение по умолчанию, а значение, которое не может быть изменено впоследствии, поэтому `DEFAULT` кажется неуместным.

Объявление константы

Между объявлениями переменных и констант существует два различия: объявление константы содержит ключевое слово `CONSTANT`, и в нем обязательно задается ее значение, которое не может быть изменено впоследствии:

```
имя CONSTANT тип_данных [NOT NULL] := | DEFAULT значение_по_умолчанию;
```

Несколько примеров объявления констант:

```
DECLARE
-- Текущий год; в течение сеанса он не будет изменяться.
l_curr_year CONSTANT PLS_INTEGER :=
    TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));

-- Использование ключевого слова DEFAULT
l_author CONSTANT VARCHAR2(100) DEFAULT 'Bill Pribyl';
```

продолжение ➤

```
-- Объявление сложного типа данных как константы.
-- Константы могут быть не только скалярами!
l_steven CONSTANT person_ot :=
    person_ot ('HUMAN', 'Steven Feuerstein', 175,
        TO_DATE ('09-23-1958', 'MM-DD-YYYY') );
```

Если в тексте явно не указано обратное, все, что говорится в этой главе о переменных, в равной степени относится и к константам.



Неименованная константа представляет собой литеральное значение — например, 2 или 'Bobby McGee'. Литерал не обладает именем, но имеет тип данных, который не объявляется, а определяется непосредственно на основании значения.

NOT NULL

Если переменной присваивается значение по умолчанию, вы также можете указать, что переменная всегда должна оставаться определенной (отличной от NULL). Для этого в объявление включается выражение `NOT NULL`. Например, следующее объявление инициализирует переменную `company_name` и указывает, что переменная всегда должна оставаться отличной от NULL:

```
company_name VARCHAR2(60) NOT NULL DEFAULT 'PCS R US';
```

При попытке выполнения следующей операции в программе будет инициировано исключение `VALUE_ERROR`:

```
company_name := NULL;
```

Кроме того, следующее объявление приводит к ошибке компиляции, так как в объявлении не указано исходное значение:

```
company_name VARCHAR2(60) NOT NULL; -- NOT NULL требует исходного значения!
```

Объявления с привязкой

При объявлении переменной тип данных очень часто задается явно:

```
l_company_name VARCHAR2(100);
```

В Oracle также существует другой метод объявления переменных, называемый *объявлением с привязкой* (anchored declaration). Он особенно удобен в тех случаях, когда значение переменной присваивается из другого источника данных, например из строки таблицы.

Объявляя «привязанную» переменную, вы устанавливаете ее тип данных на основании типа уже определенной структуры данных. Таковой может являться другая переменная PL/SQL, заранее определенный тип или подтип (`TYPE` или `SUBTYPE`), таблица базы данных либо конкретный столбец таблицы.

В PL/SQL существует два вида привязки.

Скалярная привязка. С помощью атрибута `%TYPE` переменная определяется на основании типа столбца таблицы или другой скалярной переменной PL/SQL.

Привязка к записи. Используя атрибут `%ROWTYPE`, можно определить переменную на основе таблицы или заранее определенного явного курсора PL/SQL.

Синтаксис объявления переменной с привязкой:

```
имя_переменной тип_атрибута %TYPE [необязательное_значение_по_умолчанию];
имя_переменной имя_таблицы | имя_курсора %ROWTYPE [необязательное_значение_по_
умолчанию];
```

Здесь *имя_переменной* — это имя объявляемой переменной, *тип_атрибута* — либо имя ранее объявленной переменной PL/SQL, либо спецификация столбца таблицы в формате *таблица.столбец*.

Привязка разрешается на стадии компиляции кода и не приводит к увеличению времени выполнения. Кроме того, привязка устанавливает зависимость между программным кодом и привязываемым элементом (таблицей, курсором или пакетом, содержащим переменную). Это означает, что при изменении данного элемента привязанный к нему программный код помечается как недействительный (**INVALID**). При повторной компиляции привязка выполняется заново, и таким образом код согласуется с измененным элементом.

На рис. 7.1 показано, как тип данных определяется на основе столбца таблицы базы данных и переменной PL/SQL.

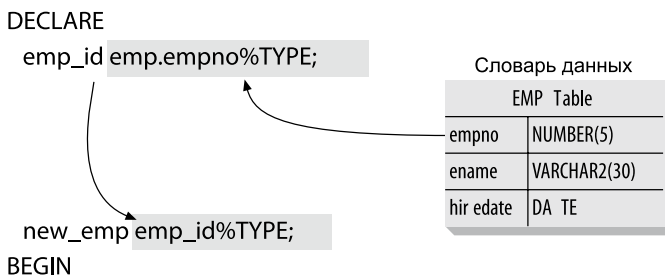


Рис. 7.1. Атрибут TYPE при объявлении с привязкой

Пример привязки переменной к столбцу таблицы:

```
l_company_id company.company_id%TYPE;
```

Аналогичным образом выполняется привязка к переменной PL/SQL; обычно это делается для того, чтобы избежать избыточных объявлений одного и того же жестко закодированного типа данных. В таких случаях в пакете обычно создается переменная, на которую затем ссылаются в программах с помощью атрибута %TYPE. (Также можно создавать в пакете подтипы SUBTYPE; эта тема рассматривается далее в этой главе.) В следующем примере приведен фрагмент кода пакета, упрощающего использование Oracle Advanced Queuing (AQ):

```
/* Файл в Сети: aq.pkg */
PACKAGE aq
IS

/* Стандартные типы данных, используемые в Oracle AQ. */
v_msgid          RAW (16);
SUBTYPE msgid_type IS v_msgid%TYPE;
v_name           VARCHAR2 (49);
SUBTYPE name_type IS v_name%TYPE;
...
END aq;
```

В пакете **aq** тип данных для хранения идентификаторов сообщений определен как RAW(16). Вместо того чтобы помнить эту спецификацию (и несколько раз жестко кодировать ее в приложении), можно объявить переменную для идентификатора сообщения следующим образом:

```
DECLARE
    my_msg_id aq.msgid_type;
BEGIN
```

Если Oracle изменит тип данных для идентификаторов сообщений, достаточно изменить определение SUBTYPE в пакете **aq**, и после перекомпиляции все объявления в программах обновятся автоматически.

Наличие объявлений с привязкой свидетельствует о том, что PL/SQL является не просто процедурным языком программирования, а разработан как расширение языка Oracle SQL. Корпорация Oracle приложила большие усилия к тому, чтобы интегрировать программные конструкции PL/SQL с базами данных, для работы с которыми используется SQL. Одно из важнейших преимуществ объявлений с привязкой заключается в том, что они позволяют писать очень гибкие приложения, которые легко адаптируются к последующим изменениям структур данных.

Привязка к курсорам и таблицам

Мы уже рассмотрели примеры объявления переменных с привязкой к столбцу базы данных и другой переменной PL/SQL. Теперь давайте посмотрим, как используется атрибут привязки %ROWTYPE.

Допустим, нам нужно выбрать одну строку из таблицы `books`. Вместо того чтобы с помощью атрибута %TYPE объявлять для каждого столбца таблицы отдельную переменную, можно воспользоваться атрибутом %ROWTYPE:

```
DECLARE
  l_book book%ROWTYPE;
BEGIN
  SELECT * INTO l_book
    FROM book
   WHERE isbn = '1-56592-335-9';
  process_book (l_book);
END;
```

Теперь предположим, что из таблицы `book` необходимо выбрать только имя автора и название книги. В этом случае мы сначала явно определим курсор, а затем на его основе объявим переменную:

```
DECLARE
  CURSOR book_cur IS
    SELECT author, title FROM book
   WHERE isbn = '1-56592-335-9';
  l_book book_cur%ROWTYPE;
BEGIN
  OPEN book_cur;
  FETCH book_cur INTO l_book; END;
```

Наконец, следующий пример демонстрирует *неявное* использование атрибута %ROWTYPE в объявлении записи `book_rec` цикла FOR:

```
BEGIN
  FOR book_rec IN (SELECT * FROM book)
  LOOP
    process_book (book_rec);
  END LOOP;
END;
```

Преимущества объявлений с привязкой

Во всех объявлениях, приводившихся в предыдущих главах книги, тип переменной (символьный, числовой, логический и т. д.) задается явно. В каждом объявлении непосредственно указывается тип данных и, как правило, ограничение, налагаемое на значение этого типа. Это распространенный подход к объявлению переменных, но в некоторых ситуациях он может вызвать проблемы.

- **Синхронизация со столбцами базы данных.** Переменная PL/SQL часто «представляет» информацию из таблицы базы данных. Если явно объявить переменную,

а затем изменить структуру таблицы, это может привести к нарушению работы программы.

- **Нормализация локальных переменных.** Допустим, переменная PL/SQL хранит вычисляемые значения, которые используются в разных местах приложения. К каким последствиям может привести повторение (жесткое кодирование) одних и тех же типов данных и ограничений во всех объявлениях?

Рассмотрим оба сценария более подробно.

Синхронизация со столбцами таблицы базы данных

В базе данных хранится информация, которая считывается и используется в приложении. Для работы с этой информацией используется как SQL, так и PL/SQL. При этом программы PL/SQL часто считывают информацию из базы данных, присваивают результат локальным переменным и после соответствующей обработки записывают информацию из переменных обратно в базу данных.

Предположим, что таблица с данными о компаниях содержит столбец `NAME` с типом данных `VARCHAR2(60)`. Локальная переменная для хранения значений этого столбца может быть объявлена следующим образом:

```
DECLARE  
  cname VARCHAR2(60);
```

Допустим, информация о компании используется в некотором приложении. В нем могут быть десятки различных процедур и отчетов, содержащих одно и то же объявление PL/SQL `VARCHAR(60)`. И все это прекрасно работает... пока не изменятся бизнес-требования или администратора базы данных не охватит жажда перемен. Тогда он запросто изменяет определение типа столбца `NAME` таблицы на `VARCHAR2(100)`, чтобы в нем помещались более длинные названия. И внезапно оказывается, что в таблицу могут попасть такие данные, что при считывании их в переменную `cname` будет инициироваться исключение `VALUE_ERROR`.

Программа становится несовместимой со структурой исходных данных. Все объявления `cname` необходимо изменить и протестировать заново — в противном случае приложение становится «миной замедленного действия», а сбой становится делом времени. Переменная, которая должна была служить локальным представлением информации из базы данных, утрачивает синхронизацию со столбцом базы данных.

Нормализация локальных переменных

Другой недостаток явного объявления типов данных проявляется при работе с переменными PL/SQL, которые содержат вычисляемые значения, не хранящиеся в базе данных. Предположим, программисты написали приложение для управления финансами компании. Во многих его программах для хранения итоговой выручки используется переменная `total_revenue`, объявленная следующим образом:

```
total_revenue NUMBER (10,2);
```

Как видите, выручка компании подсчитывается до последнего пени. В 2002 году, когда была написана спецификация приложения, максимальная выручка составляла 99 миллионов долларов, поэтому для переменной использовалось объявление `NUMBER(10,2)`. Позднее, в 2005 году, дела фирмы пошли в гору, и максимум был увеличен до `NUMBER(14,2)`. Но для этого пришлось разыскивать в приложении экземпляры переменной `total_revenue` и изменять их объявления. Работа была долгой и ненадежной — изначально была пропущена пара объявлений, и для их обнаружения пришлось проводить полное регрессионное тестирование. Одинаковые объявления были рассеяны

по всему приложению. По сути, локальные структуры данных были денормализованы с обычными последствиями для сопровождения. Жаль, что локальные переменные `total_revenue` не были объявлены со ссылкой на один тип данных, что позволило бы использовать объявления с атрибутом `%TYPE`.

Объявления с привязкой и ограничение NOT NULL

При объявлении переменной для нее можно задать ограничение `NOT NULL`, и оно будет перенесено на переменные, объявляемые на ее основе с атрибутом `%TYPE`. Если включить ограничение `NOT NULL` в объявление переменной, к которой с помощью атрибута `%TYPE` привязываются другие переменные, то для использующих ее переменных необходимо задать значение по умолчанию. Допустим, мы объявили переменную `max_available_date` с ограничением `NOT NULL`:

```
DECLARE
    max_available_date DATE NOT NULL :=
    ADD_MONTHS (SYSDATE, 3);
    last_ship_date max_available_date%TYPE;
```

Такое объявление переменной `last_ship_date` вызовет следующую ошибку компиляции:
`PLS_00218: a variable declared NOT NULL must have an initialization assignment.`

Если вы используете переменную, объявленную с ограничением `NOT NULL` в объявлении с атрибутом `%TYPE`, обязательно укажите в этом объявлении ее начальное значение. Но если источником значений переменной является столбец базы данных, объявленный с ограничением `NOT NULL`, делать это не обязательно, поскольку в подобных случаях ограничение `NOT NULL` на переменную не переносится.

Подтипы данных, определяемые программистом

PL/SQL поддерживает оператор `SUBTYPE`, который позволяет программисту определять собственные подтипы (иногда называемые абстрактными типами данных). В PL/SQL подтип представляет собой новый тип данных с тем же набором правил, но сокращенным подмножеством значений исходного типа.

Подтипы делятся на две категории.

- **Подтип с ограничениями.** Набор значений подтипа является подмножеством набора значений исходного типа данных. Например, тип `POSITIVE` является подтипом `BINARY_INTEGER` с ограничениями. В пакете `STANDARD`, где определяются типы данных и функции стандартного языка PL/SQL, подтип `POSITIVE` определяется следующим образом:

```
SUBTYPE POSITIVE IS BINARY_INTEGER RANGE 1 .. 2147483647;
```

В переменной, объявленной с типом `POSITIVE`, могут храниться только целочисленные значения больше 0.

- **Подтип без ограничений.** Этот подтип имеет такой же набор значений, что и исходный тип данных. Например, тип `FLOAT` является подтипом `NUMBER` без ограничений. Он определен в пакете `STANDARD` следующим образом:

```
SUBTYPE FLOAT IS NUMBER;
```

Подтипы объявляются в разделе объявлений анонимного блока PL/SQL, процедуры, функции или пакета. Ранее уже приводился пример объявления подтипа из пакета `STANDARD`. Общий формат объявления подтипа:

```
SUBTYPE имя_подтипа IS базовый_тип;
```

Здесь *имя_подтипа* — имя нового определяемого подтипа, а *базовый_тип* — имя типа данных, на котором основывается определяемый подтип.

Иначе говоря, подтипы данных без ограничений — это псевдонимы или альтернативные имена исходных типов данных. Несколько примеров:

```
PACKAGE utility
AS
    SUBTYPE big_string IS VARCHAR2(32767);
    SUBTYPE big_db_string IS VARCHAR2(4000);
END utility;
```

Имейте в виду, что на переменные, привязанные к подтипу, не распространяется ограничение NOT NULL, а также игнорируются начальные значения, включенные в исходное объявление переменной или спецификацию столбца.

Преобразования типов данных

В ходе выполнения программы часто возникает необходимость преобразования данных из одного типа в другой. Преобразование может выполняться двумя способами:

- **Неявно** — поиск «оптимального варианта» поручается исполнительному ядру PL/SQL;
- **Явно** — преобразование выполняется вызовом функции или соответствующим оператором PL/SQL.

В этом разделе мы сначала разберемся, как в PL/SQL выполняются неявные преобразования, а затем перейдем к изучению функций и операторов явного преобразования.

Неявное преобразование типов

Обнаружив необходимость преобразования, PL/SQL пытается привести значение к нужному типу. Вероятно, вас удивит, насколько часто это делается. На рис. 7.2 показано, какие виды неявного преобразования типов выполняются PL/SQL.

| ИЗ \ В | B | | | | | | | | | | | | | | | | | |
|-----------------------|------|----------|-------|-----------|------|-----------------------|--------|--------------|---------------|----------------|-------------|----------------|------|-----|-------|------|------|-------|
| | CHAR | VARCHAR2 | NCHAR | NVARCHAR2 | DATE | DATETIME/ INTERVAL | NUMBER | BINARY_FLOAT | BINARY_DOUBLE | BINARY_INTEGER | PLS_INTEGER | SIMPLE_INTEGER | LONG | RAW | ROWID | CLOB | BLOB | NCLOB |
| CHAR | | • | • | • | • | • | • | • | • | • | • | • | • | • | | • | • | • |
| VARCHAR 2 | • | | • | • | • | • | • | • | • | • | • | • | • | • | • | • | | • |
| NCHAR | • | • | | • | • | • | • | • | • | • | • | • | • | • | • | • | | • |
| NVARCHAR2 | • | • | • | | • | • | • | • | • | • | • | • | • | • | • | • | | • |
| DATE | • | • | • | • | | | | | | | | | | | | | | |
| DATETIME/ INTERVAL | • | • | • | • | | | | | | | | | • | | | | | |
| NUMBER | • | • | • | • | | | | • | • | • | • | • | | | | | | |
| BINARY_FLOAT | • | • | • | • | | | • | | • | • | • | • | | | | | | |
| BINARY_DOUBLE | • | • | • | • | | | • | • | | • | • | • | • | | | | | |
| BINARY_INTEGER | • | • | • | • | | | • | • | • | | • | • | • | | | | | |
| PLS_INTEGER | • | • | • | • | | | • | • | • | • | | • | • | | | | | |
| SIMPLE_INTEGER | • | • | • | • | | | • | • | • | • | • | | • | | | | | |
| LONG | • | • | • | • | | • | | | | • | • | • | | • | | • | | • |
| RAW | • | • | • | • | | | | | | | | | • | | | • | | |
| ROWID | | • | • | • | | | | | | | | | | | | | • | |
| CLOB | • | • | • | • | | | | | | | | | • | | | | | • |
| BLOB | | | | | | | | | | | | | | • | | | | |
| NCLOB | • | • | • | • | | | | | | | | | • | | | • | | |

Рис. 7.2. Неявные преобразования типов, выполняемые PL/SQL

Неявное преобразование типов осуществляется при задании в операторе или выражении литерального значения в правильном внутреннем формате, которое PL/SQL преобразует по мере необходимости. В следующем примере PL/SQL преобразует литеральную строку «125» в числовое значение 125 и присваивает его числовой переменной:

```
DECLARE
  a_number NUMBER;
BEGIN
  a_number := '125';
END;
```

Неявное преобразование типов выполняется также при передаче программе параметров не того формата, который в ней используется. В следующей процедуре таким параметром является дата. Вызывая эту процедуру, вы передаете ей строку в формате *ДД-МММ-ГГ*, которая автоматически преобразуется в дату:

```
PROCEDURE change_hiredate
  (emp_id_in IN INTEGER, hiredate_in IN DATE)

change_hiredate (1004, '12-DEC-94');
```

Неявное преобразование строки в дату выполняется в соответствии со спецификацией NLS_DATE_FORMAT. Проблема заключается в том, что в случае изменения NLS_DATE_FORMAT работа программы будет нарушена.

Ограничения неявного преобразования

Как видно из рис. 7.2, преобразование может выполняться только между определенными типами данных; PL/SQL не может преобразовать произвольный тип данных в любой другой. Более того, при некоторых неявных преобразованиях типов генерируются исключения. Возьмем следующую операцию присваивания:

```
DECLARE
  a_number NUMBER;
BEGIN
  a_number := 'abc';
END;
```

В PL/SQL нельзя преобразовать строку «abc» в число, поэтому при выполнении приведенного кода инициируется исключение VALUE_ERROR. Вы сами должны позаботиться о том, чтобы значение, для которого PL/SQL выполняет преобразование типов, могло быть конвертировано без ошибок.

Недостатки неявного преобразования

Неявное преобразование типов имеет ряд недостатков.

- PL/SQL относится к языкам со статической типизацией. Неявные преобразования означают потерю некоторых преимуществ статической типизации — таких, как ясность и надежность кода.
- Каждое неявное преобразование означает частичную потерю контроля над программой. Программист не выполняет его самостоятельно и никак им не управляет, а лишь предполагает, что оно будет выполнено и даст желаемый эффект. В этом есть элемент неопределенности — если компания Oracle изменит способ или условие выполнения преобразований, это может отразиться на программе.
- Неявное преобразование типов в PL/SQL зависит от контекста. Оно может работать в одной программе и не работать в другой, хотя на первый взгляд код кажется одинаковым. Кроме того, результат преобразования типов не всегда соответствует ожиданиями программиста.

- Программу легче читать и понять, если данные в ней преобразуются явно, поскольку при этом фиксируются различия между типами данных в разных таблицах или в таблице и коде. Исключая из программы скрытые действия, вы устраняете и потенциальную возможность ошибок.

Таким образом, в SQL и PL/SQL рекомендуется избегать неявного преобразования типов. Лучше пользоваться функциями, которые выполняют явное преобразование — это гарантирует, что результат преобразования будет точно соответствовать вашим ожиданиям.

Явное преобразование типов

Oracle предоставляет обширный набор функций и операторов, с помощью которых можно выполнить преобразование типов данных в SQL и PL/SQL. Их полный список приведен в табл. 7.1. Большая часть функций описывается в других главах книги (для них в последнем столбце указан номер главы). О функциях, которые нигде в книге не описаны, рассказывается далее в этой главе.

Таблица 7.1. Встроенные функции преобразования

| Функция | Выполняемое преобразование | Глава |
|--|---|----------|
| ASCIISTR | Строку из любого набора символов в строку ASCII из набора символов базы данных | 8 |
| CAST | Одно значение встроенного типа данных или коллекции в другой встроенный тип данных или коллекцию. Этот способ может использоваться вместо традиционных функций (таких, как TO_DATE) | 7, 9, 10 |
| CHARTOROWID | Строку в значение типа ROWID | 7 |
| CONVERT | Строку из одного набора символов в другой | 7 |
| FROM_TZ | В значение типа TIMESTAMP добавляет информацию о часовом поясе, преобразуя его тем самым в значение типа TIMESTAMP WITH TIME ZONE | 10 |
| HEXTORAW | Значение из шестнадцатеричной системы в значение типа RAW | 7 |
| MULTISET | Таблицу базы данных в коллекцию | 12 |
| NUMTODSINTERVAL | Число (или числовое выражение) в литерал INTERVAL DAY TO SECOND | 10 |
| NUMTOYMINTERVAL | Число (или числовое выражение) в литерал INTERVAL YEAR TO MONTH | 10 |
| RAWTONEX, RAWTONHEX | Значение типа RAW в шестнадцатеричный формат | 7 |
| REFTONEX | Значение типа REF в символьную строку, содержащую его шестнадцатеричное представление | 26 |
| ROWIDTOCHAR, ROWIDTONCHAR | Двоичное значение типа ROWID в символьную строку | 7 |
| TABLE | Коллекцию в таблицу базы данных; по своему действию обратна функции MULTISET | 12 |
| THE | Значение столбца в строку виртуальной таблицы базы данных | 12 |
| TO_BINARY_FLOAT | Число или строку в BINARY_FLOAT | 9 |
| TO_BINARY_DOUBLE | Число или строку в BINARY_DOUBLE | 9 |
| TO_CHAR, TO_NCHAR (числовая версия) | Число в строку (VARCHAR2 или NVARCHAR2 соответственно) | 9 |
| TO_CHAR, TO_NCHAR (версия для дат) | Дату в строку | 10 |
| TO_CHAR, TO_NCHAR (символьная версия) | Данные из набора символов базы данных в набор символов национального языка | 8 |
| TO_BLOB | Значение типа RAW в BLOB | 13 |

продолжение ⇨

Таблица 7.1 (продолжение)

| Функция | Выполняемое преобразование | Глава |
|---------------------|---|-------|
| TO_CLOB, TO_NCLOB | Значение типа VARCHAR2, NVARCHAR2 или NCLOB в CLOB (либо NCLOB) | 13 |
| TO_DATE | Строку в дату | 10 |
| TO_DSINTERVAL | Символьную строку типа CHAR, VARCHAR2, NCHAR или NVARCHAR2 в тип INTERVAL DAY TO SECOND | 10 |
| TO_LOB | Значение типа LONG в LOB | 13 |
| TO_MULTI_BYTE | Однобайтовые символы исходной строки в их многобайтовые эквиваленты (если это возможно) | 8 |
| TO_NUMBER | Строку или число (например, BINARY_FLOAT) в NUMBER | 9 |
| TO_RAW | Значение типа BLOB в RAW | 13 |
| TO_SINGLE_BYTE | Многобайтовые символы исходной строки в соответствующие однбайтовые символы | 8 |
| TO_TIMESTAMP | Символьную строку в значение типа TIMESTAMP | 10 |
| TO_TIMESTAMP_TZ | Символьную строку в значение типа TO_TIMESTAMP_TZ | 10 |
| TO_YMINTERVAL | Символьную строку типа CHAR, VARCHAR2, NCHAR или NVARCHAR2 в значение типа INTERVAL YEAR TO MONTH | 10 |
| TRANSLATE ... USING | Текст в набор символов, заданный для преобразования набора символов базы данных в национальный набор символов | 8 |
| UNISTR | Строку произвольного набора символов в Юникод | 8, 25 |

Функция CHARTOROWID

Преобразует строку типа CHAR или VARCHAR2 в значение типа ROWID. Синтаксис функции:

```
FUNCTION CHARTOROWID (исходная_строка IN CHAR) RETURN ROWID
FUNCTION CHARTOROWID (исходная_строка IN VARCHAR2) RETURN ROWID
```

Для успешного преобразования функцией CHARTOROWID строка должна состоять из 18 символов в формате:

000000ФФБББББССС

где 000000 — номер объекта данных, ФФФ — относительный номер файла базы данных, ББББББ — номер блока в файле, а ССС — номер строки в блоке. Все четыре компонента задаются в формате Base64. Если исходная строка не соответствует этому формату, инициируется исключение VALUE_ERROR.

Функция CAST

Функция CAST является очень удобным и гибким механизмом преобразования данных. Она преобразует значение любого (или почти любого) встроенного типа данных или коллекции в другой встроенный тип данных или коллекцию, и скорее всего, будет знакома всем программистам с опытом работы на объектно-ориентированных языках. С помощью функции CAST можно преобразовать неименованное выражение (число, дату, NULL и даже результат подзапроса) или именованную коллекцию (например, вложенную таблицу) в тип данных или именованную коллекцию совместимого типа.

Допустимые преобразования между встроенными типами данных показаны на рис. 7.3. Необходимо соблюдать следующие правила:

- не допускается преобразование типов данных LONG, LONG RAW, любых типов данных LOB и типов, специфических для Oracle;
- обозначению «DATE» на рисунке соответствуют типы данных DATE, TIMESTAMP, TIMESTAMP WITH TIMEZONE, INTERVAL DAY TO SECOND и INTERVAL YEAR TO MONTH;
- для преобразования именованной коллекции определенного типа в именованную коллекцию другого типа нужно, чтобы элементы обеих коллекций имели одинаковый тип;

| Из \ В | BINARY_FLOAT, BINARY_DOUBLE | CHAR, VARCHAR2 | NUMBER | DATE, TIMESTAMP, INTERVAL | RAW | ROWID, UROWID | NCHAR, NVARCHAR2 |
|---------------------------------|--------------------------------|-------------------|--------|---------------------------------|-----|------------------|---------------------|
| BINARY_FLOAT, BINARY_DOUBLE | • | • | • | | | | • |
| CHAR, VARCHAR2 | • | • | • | • | • | • | |
| NUMBER | • | • | • | | | | • |
| DATE, TIMESTAMP, INTERVAL | | • | | • | | | • |
| RAW | | • | | | • | | • |
| ROWID, UROWID | | • | | | | | |
| NCHAR, NVARCHAR2 | • | | • | | | | • |

Рис. 7.3. Преобразование встроенных типов данных

- тип UROWID не может быть преобразован в ROWID, если UROWID содержит значение ROWID индекс-таблицы.

Ниже приведен пример использования функции CAST для преобразования скалярных типов данных. Ее вызов может быть включен в SQL-команду:

```
SELECT employee_id, cast (hire_date AS VARCHAR2 (30))
FROM employee;
```

Также возможен вызов в синтаксисе PL/SQL:

```
DECLARE
    hd_display VARCHAR2 (30);
BEGIN
    hd_display := CAST (SYSDATE AS VARCHAR2);
END;
```

Намного более интересное применение CAST встречается при работе с коллекциями PL/SQL (вложенными таблицами и VARRAY), поскольку эта функция позволяет преобразовывать коллекцию из одного типа в другой. Кроме того, CAST может использоваться для работы (из инструкций SQL) с коллекцией, объявленной как переменная PL/SQL.

Обе темы подробно рассматриваются в главе 12, а следующий пример дает некоторое представление о синтаксисе и возможностях преобразования. Сначала мы создаем два типа вложенных таблиц и одну реляционную таблицу:

```
CREATE TYPE names_t AS TABLE OF VARCHAR2 (100);
CREATE TYPE authors_t AS TABLE OF VARCHAR2 (100);
CREATE TABLE favorite_authors (name VARCHAR2(200))
```

Далее пишется программа, которая связывает данные из таблицы `favorite_authors` с содержимым вложенной таблицы, объявленной и заполненной в другой программе. Рассмотрим следующий блок:

```
/* Файл в Сети: cast.sql */
1  DECLARE
2      scifi_favorites  authors_t
3      := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
4  BEGIN
5      DBMS_OUTPUT.put_line ('I recommend that you read books by:');
6
```

продолжение ⇨

```

7      FOR rec IN (SELECT column_value favs
8                  FROM TABLE (CAST (scifi_favorites AS names_t))
9                  UNION
10                 SELECT NAME
11                 FROM favorite_authors)
12      LOOP
13          DBMS_OUTPUT.put_line (rec.favs);
14      END LOOP;
15  END;
```

В строках 2 и 3 объявляется локальная вложенная таблица, заполняемая именами нескольких популярных авторов. В строках 7–11 с помощью оператора **UNION** объединяются строки таблиц **favorite_authors** и **scifi_favorites**. Для этого вложенная таблица **scifi_favorites** (локальная и не видимая для ядра SQL) *преобразуется* с использованием функции **CAST** в коллекцию типа **names_t**. Такое преобразование возможно благодаря совместимости их типов данных. После преобразования вызов команды **TABLE** сообщает ядру SQL, что вложенная таблица должна интерпретироваться как реляционная. На экран выводятся следующие результаты:

```

I recommend that you read books by:
Gene Wolfe
Orson Scott Card
Robert Harris
Sheri S. Tepper
Tom Segev
Toni Morrison
```

Функция CONVERT

Преобразует строку из одного набора символов в другой. Синтаксис функции:

```

FUNCTION CONVERT
    (исходная_строка IN VARCHAR2,
     новый_набор_символов VARCHAR2
     [, старый_набор_символов VARCHAR2])
RETURN VARCHAR2
```

Третий аргумент *старый_набор_символов* не является обязательным. Если он не задан, применяется набор символов, используемый в базе данных по умолчанию.

Функция **CONVERT** *не переводит* слова или фразы с одного языка на другой, а заменяет буквы или символы одного набора символов буквами или символами другого.

Два самых распространенных набора символов — WE8MSWIN1252 (8-разрядный набор символов Microsoft Windows, кодовая страница 1252) и AL16UTF16 (16-разрядный набор символов Юникод).

Функция HEXTORAW

Преобразует шестнадцатеричную строку типа **CHAR** или **VARCHAR2** в значение типа **RAW**. Синтаксис функции **HEXTORAW**:

```

FUNCTION HEXTORAW (исходная_строка IN CHAR) RETURN RAW
FUNCTION HEXTORAW (исходная_строка IN VARCHAR2) RETURN RAW
```

Функция RAWTONEX

Преобразует значение типа **RAW** в шестнадцатеричную строку типа **VARCHAR2**. Синтаксис функции **RAWTONEX**:

```

FUNCTION RAWTONEX (двоичное_значение IN RAW) RETURN VARCHAR2
```

Функция RAWTONEX всегда возвращает строку переменной длины, хотя обратная ей перегруженная функция HEXTORAW поддерживает оба типа строк.

Функция ROWIDTOCHAR

Преобразует двоичное значение типа ROWID в строку типа VARCHAR2. Синтаксис функции ROWIDTOCHAR:

```
FUNCTION ROWIDTOCHAR (исходная_строка IN ROWID ) RETURN VARCHAR2
```

Возвращаемая функцией строка имеет следующий формат:

OOOOOOФФБББББССС

где *OOOOOO* — номер объекта данных, *ФФФ* — относительный номер файла базы данных, *ББББББ* — номер блока в файле, а *ССС* — номер строки в блоке. Все четыре компонента задаются в формате Base64. Пример:

AAARYiAAEAAAEG8AAB

8

Строки

Переменные символьных типов предназначены для хранения текста, а для работы с ними используются символьные функции. Работа с символьными данными значительно различается по сложности от простой до весьма нетривиальной. В этой главе базовые средства PL/SQL по работе со строками рассматриваются в контексте однобайтовых наборов символов — например, тех, которые традиционно используются в Западной Европе и США. Если вы работаете в Юникоде или в другой многобайтовой кодировке или ваше приложение должно поддерживать несколько языков, обязательно ознакомьтесь с проблемами глобализации и локализации в главе 25.



Хотя типы CLOB (Character Large Object) и LONG теоретически тоже можно отнести к символьным типам, по принципам использования они отличаются от символьных типов, рассматриваемых в этой главе. Их лучше рассматривать как большие объектные типы (см. главу 13).

Строковые типы данных

Oracle поддерживает четыре строковых типа данных (табл. 8.1). Выбор типа зависит от двух факторов:

- Работаете ли вы со строками переменной или фиксированной длины?
- Хотите ли вы использовать набор символов базы данных или национальный набор символов?

| | Фиксированная длина | Переменная длина |
|-----------------------------|---------------------|------------------|
| Набор символов базы данных | CHAR | VARCHAR2 |
| Национальный набор символов | NCHAR | NVARCHAR2 |

Типы данных фиксированной длины — CHAR и NCHAR — в приложениях Oracle используются очень редко. Их вообще не рекомендуется применять, если нет особых причин работать именно со строкой фиксированной длины. Далее, в разделе «Смещение значений CHAR и VARCHAR2» рассказывается о проблемах, которые могут возникнуть при совместном использовании строковых переменных фиксированной и переменной длины (типы NCHAR и NVARCHAR2 рассматриваются в главе 25).

Тип данных VARCHAR2

В переменных типа VARCHAR2 хранятся символьные строки переменной длины. При объявлении такой строки для нее определяется максимальная длина в диапазоне от 1 до 32 767 байт. Максимальная длина может задаваться в байтах или символах, но в любом случае компилятор определяет ее в байтах. Общий синтаксис объявления VARCHAR2:

имя_переменной VARCHAR2 (*макс_длина* [CHAR | BYTE]);

Здесь *имя_переменной* — имя объявляемой переменной, *макс_длина* — ее максимальная длина, а CHAR и BYTE — аргументы, указывающие, что максимальная длина выражается в символах или в байтах соответственно.

Если максимальная длина строковой переменной VARCHAR2 задается в символах (спецификатор CHAR), то ее реальная длина в байтах вычисляется на основе максимального количества байтов, используемых для представления одного символа. Например, набор символов Юникода UTF-8 использует для представления некоторых символов до 4 байтов; следовательно, если вы работаете с UTF-8, объявление переменной типа VARCHAR2, максимальная длина которой составляет 100 символов, эквивалентно объявлению этой же переменной с максимальной длиной 300 байт.



Спецификатор длины CHAR используется в основном при работе с многобайтовыми наборами символов — такими, как UTF-8. Наборы символов и семантика операций с символьными данными рассматриваются в главе 25.

Если в объявлении переменной VARCHAR2 опустить спецификатор CHAR или BYTE, тогда заданное значение длины будет интерпретировано в байтах или символах в зависимости от параметра инициализации NLS_LENGTH_SEMANTICS. Текущее значение можно узнать, обратившись с запросом к NLS_SESSION_PARAMETERS. Несколько примеров объявления строк типа VARCHAR2:

```
DECLARE
  small_string VARCHAR2(4);
  line_of_text VARCHAR2(2000);
  feature_name VARCHAR2(100 BYTE); -- Строка длиной 100 байт
  emp_name VARCHAR2(30 CHAR); ----- Строка длиной 30 символов
```

Итак, максимальная длина переменной типа VARCHAR2 в PL/SQL составляет 32 767 байт. Это ограничение действует независимо от того, определяется ли длина строки в байтах или символах. До выхода версии 12c максимальная длина типа данных VARCHAR2 в SQL была равна 4000; в 12c она была увеличена до максимума PL/SQL: 32 767 байт. Однако следует учитывать, что SQL поддерживает этот максимум только в том случае, если параметру инициализации MAX_SQL_STRING_SIZE задано значение EXTENDED; по умолчанию используется значение STANDARD.

Если вам понадобится работать со строками длиной более 4000 байт, рассмотрите возможность их хранения в столбцах типа CLOB. О столбцах этого типа подробно рассказано в главе 13.

Тип данных CHAR

Тип данных CHAR определяет строку фиксированной длины. При объявлении такой строки необходимо задать ее максимальную длину в диапазоне от 1 до 32 767 байт. Длина может задаваться как в байтах, так и в символах. Например, следующие два объявления создают строки длиной 100 байт и 100 символов соответственно:

```
feature_name CHAR(100 BYTE);
feature_name CHAR(100 CHAR);
```

Реальный размер 100-символьной строки в байтах зависит от текущего набора символов базы данных. Если используется набор символов с переменной длиной кодировки, PL/SQL выделяет для строки столько места, сколько необходимо для представления заданного количества символов с максимальным количеством байтов. Например, в наборе UTF-8, где символы имеют длину от 1 до 4 байт, PL/SQL при создании строки для хранения 100 символов зарезервирует 300 байт (3 байта × 100 символов).

Мы уже знаем, что при отсутствии спецификатора CHAR или BYTE результат будет зависеть от параметра NLS_LENGTH_SEMANTICS. При компиляции программы эта настройка сохраняется вместе с ней и может использоваться повторно или заменяться при последующей перекомпиляции. (Параметры компиляции рассматриваются в главе 20.) С настройкой по умолчанию для следующего объявления будет создана строка длиной 100 байт:

```
feature_name CHAR(100);
```

Если длина строки не указана, PL/SQL объявит строку длиной 1 байт. Предположим, переменная объявляется так:

```
feature_name CHAR;
```

Как только этой переменной присваивается строка длиной более одного символа, PL/SQL инициирует универсальное исключение VALUE_ERROR. Но при этом не указывается, где именно возникла проблема. Если эта ошибка была получена при объявлении новых переменных или констант, проверьте свои объявления на небрежное использование CHAR. Чтобы избежать проблем и облегчить работу программистов, которые придут вам на смену, *всегда* указывайте длину строки типа CHAR. Несколько примеров:

```
yes_or_no CHAR (1) DEFAULT 'Y';
line_of_text CHAR (80 CHAR); ----- Всегда все 80 символов!
whole_paragraph CHAR (10000 BYTE); -- Подумайте обо всех этих пробелах...
```

Поскольку строка типа CHAR имеет фиксированную длину, PL/SQL при необходимости дополняет справа присвоенное значение пробелами, чтобы фактическая длина соответствовала максимальной, указанной в объявлении.

До выхода версии 12c максимальная длина типа данных CHAR в SQL была равна 2000; в 12c она была увеличена до максимума PL/SQL: 32 767 байт. Однако следует учитывать, что SQL поддерживает этот максимум только в том случае, если параметру инициализации MAX_SQL_STRING_SIZE задано значение EXTENDED.

Строковые подтипы

PL/SQL поддерживает некоторые строковые подтипы (табл. 8.1), которые тоже могут использоваться для объявления символьных строк. Многие из этих подтипов определены только для обеспечения совместимости со стандартом ANSI SQL. Вряд ли они вам когда-нибудь понадобятся, но знать о них все же нужно.

Каждый из перечисленных в таблице подтипов эквивалентен одному из базовых типов данных PL/SQL, указанных в правом столбце. Например:

```
feature_name VARCHAR2(100);
feature_name CHARACTER VARYING(100);
feature_name CHAR VARYING(100);
feature_name STRING(100);
```

Подтип VARCHAR заслуживает особого внимания. Уже на протяжении нескольких лет корпорация Oracle собирается изменить определение подтипа данных VARCHAR (в результате чего он перестанет быть эквивалентным VARCHAR2) и предупреждает, что пользоваться им не следует. Я согласен с этой рекомендацией: если существует опасность, что Oracle

(или комитет ANSI) изменит поведение VARCHAR, неразумно полагаться на его поведение. Используйте вместо него VARCHAR2.

Таблица 8.1. Строковые подтипы и эквивалентные типы данных PL/SQL

| Подтип | Эквивалентный тип |
|----------------------------|-------------------|
| CHAR VARYING | VARCHAR2 |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR2 |
| NATIONAL CHAR | NCHAR |
| NATIONAL CHAR VARYING | NVARCHAR2 |
| NATIONAL CHARACTER | NCHAR |
| NATIONAL CHARACTER VARYING | NVARCHAR2 |
| NCHAR VARYING | NVARCHAR2 |
| STRING | VARCHAR2 |
| VARCHAR | VARCHAR2 |

О работе со строками

Работа со строками в основном сводится к вызову многочисленных встроенных строковых функций Oracle, поэтому авторы рекомендуют хотя бы в общих чертах познакомиться с функциями, предлагаемыми Oracle. В следующих подразделах вы узнаете, как записывать строковые константы, а также познакомитесь с важнейшими строковыми функциями.

Определение строковых констант

Один из способов загрузки строковых данных в программу PL/SQL заключается в выполнении инструкции `SELECT`, возвращающей значения строкового типа. Также возможно определение строковых констант непосредственно в коде. Строковые константы заключаются в одинарные кавычки:

```
'Brighten the corner where you are.'
```

Если одинарную кавычку потребуется включить в строковую константу, ее можно продублировать:

```
'Aren''t you glad you''re learning PL/SQL with O''Reilly?'
```

Другое, более элегантное решение основано на определении пользовательских ограничителей строк. Для этой цели используется префикс `q` (также допускается использование символа `Q` верхнего регистра). Пример:

```
q'!Aren't you glad you're learning PL/SQL with O'Reilly?!'
```

или:

```
q'{'Aren't you glad you're learning PL/SQL with O'Reilly?}'
```

При использовании префикса `q` вся строка должна быть заключена в одиночные кавычки. Символ, следующий за первой кавычкой (восклицательный знак в первом из двух примеров), становится *ограничителем* строки. Таким образом, первая строка с префиксом `q` состоит из всех символов между двумя восклицательными знаками.



Если начальным ограничителем является символ `[`, `{`, `<` или `(`, то конечным ограничителем должен быть символ `]`, `}`, `>` или `)` соответственно.

Как правило, в представлении строковых констант используется набор символов базы данных. Если константа присваивается переменной `NCHAR` или `NVARCHAR2`, она неявно преобразуется в символы национального набора (см. главу 25). Oracle выполняет такие преобразования по мере необходимости, и программисту об этом заботиться не нужно. Если же потребуется явно указать, что строковая константа задается в национальном наборе символов, поставьте перед ней префикс `n`:

```
n'Pils vom faß: 1€'
```

Чтобы при определении строки в национальном наборе некоторые символы задавались числовыми кодами Юникода, используйте префикс `u`:

```
u'Pils vom fa\00DF: 1\20AC'
```

Здесь `00DF` — код немецкой буквы «ß», а `20AC` — код знака евро. Полученная константа не отличается от предшествующего примера с префиксом `n`.

Значение строковой константы можно сохранить в переменной оператором присваивания:

```
DECLARE
    jonathans_motto VARCHAR2(50);
BEGIN
    jonathans_motto := 'Brighten the corner where you are.';
END;
```

Константы также можно передавать встроенным функциям. Например, следующий вызов функции `LENGTH` определяет количество символов в переданной строке:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(
        LENGTH('Brighten the corner where you are.')
    );
END;
```

При выполнении этого кода мы узнаем, что строка состоит из 34 символов.

При выполнении кода PL/SQL в `SQL*Plus` или `SQL Developer` часто возникают проблемы с символом `&`. Дело в том, что в этих программах указанный символ используется в качестве префикса при замене, и, встретив его, `SQL*Plus` запрашивает значение переменной. Например:

```
SQL> BEGIN
2     DBMS_OUTPUT.PUT_LINE ('Generating & saving test data.');
```

```
3 END;
```

```
4 /
```

Enter value for saving:

У проблемы существует несколько решений. Одно из них, хорошо подходящее для `SQL*Plus` и `SQL Developer`, заключается в выполнении команды `SET DEFINE OFF`, отключающей подстановку переменных. Другие решения приводятся в книге Джонатана Генника «Oracle SQL*Plus: The Definitive Guide» (издательство O'Reilly).

Непечатаемые символы

Встроенная функция `CHR` особенно удобна в тех случаях, когда в программный код необходимо включить ссылку на непечатаемый символ. Допустим, вы строите отчет, в котором выводятся адреса компаний. Помимо строк с названиями города, страны и индекса адрес может содержать до четырех дополнительных строк, и значение каждой из них должно выводиться с новой строки. Все строки адреса можно объединить в одно длинное текстовое значение и использовать функцию `CHR` для вставки разрывов строк в нужных местах. В стандартной кодировке ASCII символ новой строки имеет код 10, поэтому программа может выглядеть так:

```

SELECT name || CHR(10)
       || address1 || CHR(10)
       || address2 || CHR(10)
       || city || ', ' || state || ' ' || zipcode
       AS company_address
FROM company

```

Предположим, в таблицу была вставлена следующая строка:

```

BEGIN
  INSERT INTO company
    VALUES ('Harold Henderson',
            '22 BUNKER COURT',
            NULL,
            'WYANDANCH',
            'MN',
            '66557');

  COMMIT;
END;
/

```

Вывод будет выглядеть примерно так:

```

COMPANY_ADDRESS
-----
Harold Henderson
22 BUNKER COURT

WYANDANCH, MN 66557

```



Символ с кодом 10 обозначает разрыв строки в системах Linux и Unix. В Windows для этой цели используется комбинация символов CHR(12) || CHR(10). Возможно, в других средах вам придется использовать какие-то другие символы.

Не хотите, чтобы в выходных данных присутствовали пустые строки? Нет проблем. Задача легко решается умным использованием функции NVL2:

```

SELECT name
       || NVL2(address1, CHR(10) || address1, '')
       || NVL2(address2, CHR(10) || address2, '')
       || CHR(10) || city || ', ' || state || ' ' || zipcode
       AS company_address
FROM company

```

Теперь запрос возвращает один отформатированный столбец на компанию. Функция NVL2 возвращает третий аргумент, если первый аргумент равен NULL, или второй аргумент во всех остальных случаях. В данном примере, если значение address1 равно NULL, возвращается пустая строка (''); то же самое происходит с остальными столбцами адресов. В результате пустая строка исчезает из адреса:

```

COMPANY_ADDRESS
-----
Harold Henderson
22 BUNKER COURT
WYANDANCH, MN 66557

```

Функция ASCII фактически является обратной по отношению к CHR: она возвращает десятичное представление заданного символа в наборе символов базы данных. Например, следующий фрагмент выводит десятичный код буквы «J»:

```

BEGIN
  DBMS_OUTPUT.PUT_LINE(ASCII('J'));
END;

```

Как выясняется, буква «J» (по крайней мере в кодировке UTF-8) представлена кодом 74.



Интересные примеры использования функции CHR приводятся далее, в разделе «Традиционный поиск и замена».

Конкатенация строк

Существует два способа объединения строк: функция `CONCAT` и оператор конкатенации, представленный двумя вертикальными чертами `||`. В практическом программировании намного чаще применяется оператор конкатенации. Для чего нужны два механизма, спросите вы? Дело в том, что при трансляции кода между серверами, использующими кодировки ASCII и EBCDIC, могут возникнуть проблемы с символами вертикальной черты, а на некоторых клавиатурах для ввода этих символов приходится проявлять настоящие чудеса ловкости. Если вам неудобно работать с вертикальными чертами, используйте функцию `CONCAT`, которая получает два аргумента:

`CONCAT (строка1, строка2)`

Функция `CONCAT` всегда присоединяет *строку2* в конец *строки1* и возвращает результат. Если одна из двух строк равна `NULL`, функция `CONCAT` возвращает отличный от `NULL` аргумент. Если обе строки равны `NULL`, то `CONCAT` возвращает `NULL`. Если входные строки не относятся к типу `CLOB`, итоговая строка будет относиться к типу `VARCHAR2`. Если одна или обе входные строки относятся к типу `CLOB`, то в результате конкатенации будет получен тип `CLOB`. Если одна из строк относится к типу `NCLOB`, итоговая строка тоже относится к типу `NCLOB`. В общем случае возвращаемое значение относится к типу, сохраняющему максимум полезной информации. Несколько примеров использования `CONCAT`:

```
CONCAT ('abc', 'defg') --> 'abcdefg'
CONCAT (NULL, 'def') --> 'def'
CONCAT ('ab', NULL) --> 'ab'
CONCAT (NULL, NULL) --> NULL
```

Обратите внимание: функция позволяет объединять только две строки, а оператор конкатенации — сразу несколько строк. Пример:

```
DECLARE
  x VARCHAR2(100);
BEGIN
  x := 'abc' || 'def' || 'ghi';
  DBMS_OUTPUT.PUT_LINE(x);
END;
```

Результат:

```
abcdefghi
```

Для выполнения аналогичной конкатенации с использованием `CONCAT` приходится задействовать вложенные вызовы:

```
x := CONCAT(CONCAT('abc', 'def'), 'ghi');
```

Как видите, оператор `||` не только удобнее `CONCAT`, но и программный код получается намного более понятным.

Преобразование регистра

Регистр символов часто играет важную роль в работе со строками. Например, может потребоваться сравнить две строки независимо от регистра. Выбор решения этой проблемы зависит как от версии базы данных, так и от области действия выполняемых операций.

Преобразование строки к верхнему или нижнему регистру

Для решения проблем с регистром символов можно воспользоваться встроенной функцией UPPER или LOWER. Эти функции преобразуют всю строку за одну операцию. Пример:

```
DECLARE
  name1 VARCHAR2(30) := 'Andrew Sears';
  name2 VARCHAR2(30) := 'ANDREW SEARS';
BEGIN
  IF LOWER(name1) = LOWER(name2) THEN
    DBMS_OUTPUT.PUT_LINE('Имена совпадают.');
```

```
  END IF;
```

```
END;
```

В этом примере обе строки обрабатываются функцией LOWER, так что в итоговом сравнении участвуют строки 'andrew sears' и 'andrew sears'.

Сравнение без учета регистра символов

Начиная с Oracle10g Release 2, появилась возможность использования параметров инициализации NLS_COMP и NLS_SORT для включения режима сравнения без учета регистра. Задайте параметру NLS_COMP значение LINGUISTIC; тем самым вы прикажете Oracle использовать NLS_SORT для сравнений строк. Затем задайте параметру NLS_SORT значение, соответствующее сравнению без учета регистра — например, BINARY_CI или XWEST_EUROPEAN_CI. (Суффикс _CI означает «Case Insensitivity», то есть «Без учета регистра».) Приведенный далее простой пример показывает, какие проблемы решаются при помощи NLS_COMP. Требуется взять список имен и определить, какое из них должно стоять на первом месте:

```
SELECT LEAST ('JONATHAN', 'Jonathan', 'jon') FROM dual
```

В моей системе этот вызов LEAST возвращает строку 'JONATHAN'. Дело в том, что в порядке сортировки символы верхнего регистра предшествуют символам нижнего регистра. По умолчанию параметру NLS_COMP задается значение BINARY, при котором результат строковых сравнений, выполняемых такими функциями, как LEAST, определяется кодами символов.

Возможно, вы предпочитаете, чтобы функция LEAST игнорировала регистр символов и возвращала 'jon' вместо 'JONATHAN'. Измените значение NLS_COMP, чтобы при сортировке учитывалось значение NLS_SORT:

```
ALTER SESSION SET NLS_COMP=LINGUISTIC
```

Теперь необходимо изменить NLS_SORT с указанием нужных правил сортировки.

По умолчанию переменная NLS_SORT часто равна BINARY, но значение может быть и другим в зависимости от конфигурации системы. В нашем примере будет использоваться сортировка BINARY_CI:

```
ALTER SESSION SET NLS_SORT=BINARY_CI
```

Попробуем снова вызвать LEAST:

```
SELECT LEAST ('JONATHAN', 'Jonathan', 'jon') FROM dual
```

На этот раз будет получен результат 'jon'. Пример кажется простым, но добиться такого результата без только что описанной сортировки по правилам языка будет нелегко.

Действие языковой сортировки распространяется не только на функции, но и на простые сравнения строк. Пример:

```
BEGIN
  IF 'Jonathan' = 'JONATHAN' THEN
    DBMS_OUTPUT.PUT_LINE('It is true!');
```

```
  END IF;
```

```
END;
```

При указанных значениях параметров NLS_COMP и NLS_SORT выражение 'Jonathan' = 'JONATHAN' в этом примере равно TRUE.



Параметры NLS_COMP и NLS_SORT влияют на все операции со строками. Они продолжают действовать вплоть до их явного изменения или до завершения сеанса.

Oracle также поддерживает режим сортировки без учета диакритических знаков; чтобы включить его, присоедините к имени правила сортировки суффикс _AI (вместо _CI). За полным списком правил языковой сортировки обращайтесь к документации *Oracle Database Globalization Support Guide*. В этом руководстве также подробно объясняется действие параметров NLS_COMP и NLS_SORT. Дополнительная информация о параметрах NLS также представлена в главе 25.

Регистр символов и индексы

При работе со строками часто возникает необходимость в поиске и сравнении без учета регистра символов. Но если вы реализуете описанный здесь прием, вдруг выясняется, что ваше приложение перестает использовать индексы и начинает работать слишком медленно. Будьте внимательны, чтобы ваши действия не повредили использованию индексов в SQL. Для наглядности рассмотрим пример с демонстрационной таблицей hr.employees. Таблица employees использует индекс emp_name_ix для столбцов last_name, first_name. Мой код включает следующую команду SQL:

```
SELECT * FROM employees WHERE last_name = lname
```

Изначально код использует индекс emp_name_ix, но когда я задаю параметры NLS_COMP=LINGUISTIC и NLS_SORT=BINARY_CI, чтобы включить поиск без учета регистра, индекс перестает использоваться, и операции выполняются с полным просмотром таблицы — со всеми вытекающими последствиями! Одно из возможных решений заключается в использовании индекса на базе функции, игнорирующего регистр символов:

```
CREATE INDEX last_name_ci ON EMPLOYEES (NLSSORT(last_name, 'NLS_SORT=BINARY_CI'))
```

Теперь при выполнении запросов без учета регистра символов используется индекс без учета регистра, а быстродействие программы остается на высоком уровне.

Преобразование первого символа к верхнему регистру

Кроме функций UPPER и LOWER, для преобразования регистра символов используется функция INITCAP. Она преобразует первую букву каждого слова в строке к верхнему регистру, а все остальные буквы — к нижнему регистру. Например, следующий фрагмент:

```
DECLARE
  name VARCHAR2(30) := 'MATT williams';
BEGIN
  DBMS_OUTPUT.PUT_LINE(INITCAP(name));
END;
```

выводит следующий результат:

```
Matt Williams
```

Идея использовать INITCAP для форматирования имен выглядит заманчиво, и все будет хорошо, пока вы не столкнетесь с особым случаем:

```
DECLARE
  name VARCHAR2(30) := 'JOE mcwilliams';
BEGIN
  DBMS_OUTPUT.PUT_LINE(INITCAP(name));
END;
```

Результат:

Joe McWilliams

Правильное написание фамилии — «McWilliams», а не «Mcwilliams». Помните, что функция INITCAP временами удобна, но она выдает неверный результат для имен и слов, которые содержат более одной буквы в верхнем регистре.

Традиционный поиск и замена

Достаточно часто в строке требуется найти заданный фрагмент текста. Начиная с Oracle10g, для подобных манипуляций с текстом могут использоваться регулярные выражения; данная возможность подробно описана позднее в этой главе. Если вы еще не перешли на Oracle10g или более позднюю версию, существует решение, совместимое со старыми версиями базы данных. Функция INSTR возвращает позицию подстроки в большей строке. Следующий фрагмент находит позиции всех запятых в списке имен:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
  comma_location NUMBER := 0;
BEGIN
  LOOP
    comma_location := INSTR(names,',',comma_location+1);
    EXIT WHEN comma_location = 0;
    DBMS_OUTPUT.PUT_LINE(comma_location);
  END LOOP;
END;
```

Результат:

```
5
10
14
21
28
34
```

Первый аргумент INSTR содержит строку, в которой проводится поиск. Второй аргумент задает искомую подстроку — в данном случае это запятая. Третий аргумент задает позицию, в которой начинается поиск. После каждой найденной запятой цикл продолжает поиск с символа, следующего за найденным. Если совпадения отсутствуют, INSTR возвращает ноль, и цикл на этом завершается.

Следующий естественный шаг после обнаружения текста в строке — его извлечение. Естественно, нас интересуют не запятые, а те имена, которые они разделяют. Для извлечения строковых данных применяется функция SUBSTR:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
  names_adjusted VARCHAR2(61);
  comma_location NUMBER := 0;
  prev_location NUMBER := 0;
BEGIN
  -- Включение запятой за последним именем
  names_adjusted := names || ',';
  LOOP
    comma_location := INSTR(names_adjusted,',',comma_location+1);
    EXIT WHEN comma_location = 0;
    DBMS_OUTPUT.PUT_LINE(
      SUBSTR(names_adjusted,
        prev_location+1,
        comma_location-prev_location-1));
    prev_location := comma_location;
  END LOOP;
END;
```

Программа выдает следующий список имен:

```
Anna
Matt
Joe
Nathan
Andrew
Aaron
Jeff
```

В приведенном коде следует обратить внимание на два ключевых момента. Во-первых, в конец строки добавляется запятая, упрощающая реализацию логики цикла. Теперь за каждым именем в `names_adjusted` следует запятая; это упрощает нашу задачу. Во-вторых, каждый раз, когда цикл доходит до вызова `DBMS_OUTPUT.PUT_LINE`, в переменных `prev_location` и `comma_location` сохраняются позиции символов по обе стороны от выводимого имени. В этом случае все сводится к простым вычислениям и вызову функции `SUBSTR`, которая получает три аргумента:

- `names_adjusted` — строка, из которой извлекается имя;
- `prev_location+1` — позиция первой буквы имени. Вспомните, что в переменной `prev_location` хранится позиция символа, непосредственно предшествующего выводимому имени (обычно запятой). Эта позиция увеличивается на единицу.
- `comma_location-prev_location-1` — количество извлекаемых символов. Вычитание единицы предотвращает вывод завершающей запятой.

ОТРИЦАТЕЛЬНЫЕ ПОЗИЦИИ В СТРОКАХ

Некоторые встроенные строковые функции Oracle, прежде всего `SUBSTR` и `INSTR`, позволяют задать начальную позицию для извлечения или поиска в обратном направлении от конца строки. Например, для извлечения последних 10 символов строки можно воспользоваться следующей командой:

```
SUBSTR('Brighten the corner where you are',-10)
```

Функция возвращает строку «`ge you are`». Обратите внимание на то, что в качестве начальной позиции указывается значение `-10`.

Задавая отрицательную начальную позицию, вы приказываете `SUBSTR` вести отсчет в обратном направлении от конца строки.

`INSTR` добавляет интересный нюанс: если задать отрицательный начальный индекс, функция `INSTR`:

1. Отсчитывает указанное количество символов от конца строки, чтобы определить, откуда следует начать поиск.
2. Проводит поиск в обратном направлении от заданной позиции к началу строки.

Шаг 1 работает так же, как и у `SUBSTR`, но на шаге 2 перемещение происходит в обратном направлении. Например, следующая команда ищет второе вхождение «`ge`» от конца строки:

```
INSTR('Brighten the corner where you are','re',-1,2)
```

Чтобы вам было проще понять смысл происходящего, обозначим позиции букв в строке:

```
111111111112222222223333
123456789012345678901234567890123
INSTR('Brighten the corner where you are','re',-1,2)
```

Результат равен 24. Четвертый параметр — 2 — запрашивает второе вхождение подстроки «`ge`». Третий параметр равен `-1`, так что поиск начинается от последнего символа строки (символа, непосредственно предшествующего закрывающей кавычке). Поиск ведется от конца к началу, от «`ge`» в конце «`are`» (первое вхождение) и до «`ge`» в конце «`where`».

Все эти операции поиска-замены утомительны. Иногда сложность кода удается сократить за счет нетривиального использования некоторых встроенных функций. Давайте воспользуемся функцией `REPLACE` для замены всех запятых символами новой строки:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    REPLACE(names, ',', chr(10))
  );
END;
```

Результат:

```
Anna
Matt
Joe
Nathan
Andrew
Aaron
Jeff
```

Функция `REPLACE` позволила избавиться от циклического перебора. Код, приводящий к тем же результатам, получился более простым и элегантным. Конечно, такая замена возможна не всегда, и все же не стоит забывать об альтернативных решениях. В программировании всегда существует несколько путей к желаемому результату!

Дополнение

Иногда требуется, чтобы обрабатываемая строка имела фиксированный размер. Функции `LPAD` и `RPAD` дополняют строку пробелами (или другие символы) слева или справа до заданной длины. В следующем примере эти две функции выводят список имен в два столбца, причем левый столбец выравнивается по левому краю, а правый — по правому краю:

```
DECLARE
  a VARCHAR2(30) := 'Jeff';
  b VARCHAR2(30) := 'Eric';
  c VARCHAR2(30) := 'Andrew';
  d VARCHAR2(30) := 'Aaron';
  e VARCHAR2(30) := 'Matt';
  f VARCHAR2(30) := 'Joe';
BEGIN
  DBMS_OUTPUT.PUT_LINE( RPAD(a,10) || LPAD(b,10) );
```

Результат:

```
Jeff          Eric
Andrew       Aaron
Matt         Joe
```

По умолчанию строки дополняются пробелами. При желании можно задать используемый символ в третьем аргументе. Приведите строки кода к следующему виду:

```
DBMS_OUTPUT.PUT_LINE( RPAD(a,10,'.') || LPAD(b,10,'.') );
DBMS_OUTPUT.PUT_LINE( RPAD(c,10,'.') || LPAD(d,10,'.') );
DBMS_OUTPUT.PUT_LINE( RPAD(e,10,'.') || LPAD(f,10,'.') );
```

Результат будет выглядеть так:

```
Jeff.....Eric
Andrew.....Aaron
Matt.....Joe
```

В качестве заполнителя даже может использоваться целая строка:

```
DBMS_OUTPUT.PUT_LINE( RPAD(a,10,'---') || LPAD(b,10,'---') );
DBMS_OUTPUT.PUT_LINE( RPAD(c,10,'---') || LPAD(d,10,'---') );
DBMS_OUTPUT.PUT_LINE( RPAD(e,10,'---') || LPAD(f,10,'---') );
```

Новая версия результата:

```
Jeff-----Eric
Andrew-----Aaron
Matt-----Joe
```

Символы или строки-заполнители размещаются слева направо — всегда, даже при использовании функции `RPAD`. Чтобы убедиться в этом, достаточно повнимательнее присмотреться к 10-символьному «столбцу» с именем `Joe`.

При использовании `LPAD` и `RPAD` необходимо учитывать возможность того, что длина некоторых входных строк изначально может оказаться больше нужной ширины (или равной ей). Для примера попробуем уменьшить ширину столбца до четырех символов:

```
DBMS_OUTPUT.PUT_LINE(   RPAD(a,4) || LPAD(b,4)   );
DBMS_OUTPUT.PUT_LINE(   RPAD(c,4) || LPAD(d,4)   );
DBMS_OUTPUT.PUT_LINE(   RPAD(e,4) || LPAD(f,4)   );
```

Теперь результат будет выглядеть так:

```
JeffEric
AndrAaro
Matt Joe
```

Обратите особое внимание на вторую строку: имена «Andrew» и «Aaron» усечены до четырех символов.

Усечение строк

Действие функций `TRIM`, `LTRIM` и `RTRIM` противоположно действию `LPAD` и `RPAD`: эти функции удаляют символы от начала и от конца строки. Пример:

```
DECLARE
  a VARCHAR2(40) := 'This sentence has too many periods.....';
  b VARCHAR2(40) := 'The number 1';
BEGIN
  DBMS_OUTPUT.PUT_LINE(   RTRIM(a, '.')   );
  DBMS_OUTPUT.PUT_LINE(
    LTRIM(b, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz')
  );
END;
```

Результат:

```
This sentence has too many periods
1
```

Как видите, функция `RTRIM` удалила все точки. Второй аргумент этой функции определяет удаляемые символы. В приведенном примере вызов `LTRIM` выглядит немного абсурдно, но он демонстрирует возможность задания целого набора удаляемых символов. Мы запрашиваем удаление всех букв и пробелов от начала строки `b` и получаем желаемое.

По умолчанию функции удаляют пробелы в начале или в конце строки. Вызов `RTRIM(a)` эквивалентен `RTRIM(a, ' ')`. То же самое относится к `LTRIM(a)` и `LTRIM(a, ' ')`.

Еще одна функция удаления символов из строки — `TRIM` — была включена в Oracle8i для более полного соответствия стандарту ISO SQL. Как видно из следующего примера, функция `TRIM` несколько отличается от `LTRIM` и `RTRIM`:

```
DECLARE
  x VARCHAR2(30) := '.....Hi there!.....';
BEGIN
  DBMS_OUTPUT.PUT_LINE(   TRIM(LEADING '.' FROM x)   );
  DBMS_OUTPUT.PUT_LINE(   TRIM(TRAILING '.' FROM x)   );
  DBMS_OUTPUT.PUT_LINE(   TRIM(BOTH '.' FROM x)   );
  --По умолчанию удаление производится с обоих концов строки
  DBMS_OUTPUT.PUT_LINE(   TRIM('.') FROM x   );
```

```
--По умолчанию удаляются пробелы:
DBMS_OUTPUT.PUT_LINE(  TRIM(x)  );
END;
```

Результат:

```
Hi there!.....
.....Hi there!
Hi there!
Hi there!
.....Hi there!.....
```

Одна функция позволяет усекать строку с любой из двух сторон или же с обеих сторон сразу. Тем не менее вы можете задать только один удаляемый символ. Например, следующая запись недопустима:

```
TRIM(BOTH '.,;' FROM x)
```

Вместо нее для решения нашей конкретной задачи приходится использовать комбинацию RTRIM и LTRIM:

```
RTRIM(LTRIM(x, '.,;'), '.,;')
```

Поиск и замена с использованием регулярных выражений

В Oracle10g в области работы со строками произошли очень серьезные изменения: была реализована поддержка регулярных выражений. Причем речь идет не об упрощенной поддержке регулярных выражений вроде предиката LIKE, которая встречается в других СУБД. Компания Oracle предоставила в наше распоряжение отлично проработанный, мощный набор функций — то, что было необходимо в PL/SQL.

Регулярные выражения образуют своего рода язык для описания и обработки текста. Читатели, знакомые с языком Perl, уже разбираются в этой теме, поскольку Perl способствовал распространению регулярных выражений в большей степени, чем любой другой язык. Поддержка регулярных выражений в Oracle10g довольно близко соответствовала стандарту регулярных выражений POSIX (Portable Operating System Interface). В Oracle10g Release 2 появилась поддержка многих нестандартных, но весьма полезных операторов из мира Perl, а в Oracle11g возможности регулярных выражений были дополнительно расширены.

Проверка наличия совпадения

Регулярные выражения используются для описания текста, который требуется найти в строке (и возможно, подвергнуть дополнительной обработке). Давайте вернемся к примеру, который приводился ранее в разделе «Традиционный поиск и замена»:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
```

Допустим, мы хотим определить на программном уровне, содержит ли строка список имен, разделенных запятыми. Для этого мы воспользуемся функцией REGEXP_LIKE, обнаруживающей совпадения шаблона в строке:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
  names_adjusted VARCHAR2(61);
  comma_delimited BOOLEAN;
BEGIN
  --Поиск по шаблону
  comma_delimited := REGEXP_LIKE(names, '^[a-z A-Z]*,)+([a-z A-Z]*){1}$');

  --Вывод результата
```

продолжение ➤

```
DBMS_OUTPUT.PUT_LINE(
CASE comma_delimited
WHEN true THEN 'Обнаружен список с разделителями!'
ELSE 'Совпадение отсутствует.'
END);
```

END;

Результат:

Обнаружен список с разделителями

Чтобы разобраться в происходящем, необходимо начать с выражения, описывающего искомый текст. Общий синтаксис функции REGEXP_LIKE выглядит так:

REGEXP_LIKE (*исходная_строка*, *шаблон* [, *модификаторы*])

Здесь *исходная_строка* — символьная строка, в которой ищутся совпадения; *шаблон* — регулярное выражение, совпадения которого ищутся в *исходной_строке*; *модификаторы* — один или несколько модификаторов, управляющих процессом поиска. Если функция REGEXP_LIKE находит совпадение *шаблона* в *исходной_строке*, она возвращает логическое значение TRUE; в противном случае возвращается FALSE.

Процесс построения регулярного выражения выглядел примерно так:

○ [a-z A-Z]

Каждый элемент списка имен может состоять только из букв и пробелов. Квадратные скобки определяют набор символов, которые могут входить в совпадение. Диапазон a–z описывает все буквы нижнего регистра, а диапазон A–Z — все буквы верхнего регистра. Пробел находится между двумя компонентами выражения. Таким образом, этот шаблон описывает один любой символ нижнего или верхнего регистра или пробел.

○ [a-z A-Z]*

Звездочка является квантификатором — служебным символом, который указывает, что каждый элемент списка содержит ноль или более повторений совпадения, описанного шаблоном в квадратных скобках.

○ [a-z A-Z]*,

Каждый элемент списка должен завершаться запятой. Последний элемент является исключением, но пока мы не будем обращать внимания на эту подробность.

○ ([a-z A-Z]*,)

Круглые скобки определяют подвыражение, которое описывает некоторое количество символов, завершаемых запятой. Мы определяем это подвыражение, потому что оно должно повторяться при поиске.

○ ([a-z A-Z]*,)+

Знак + — еще один квантификатор, применяемый к предшествующему элементу (то есть к подвыражению в круглых скобках). В отличие от * знак + означает «одно или более повторений». Список, разделенный запятыми, состоит из одного или нескольких повторений подвыражения.

○ ([a-z A-Z]*,)+([a-z A-Z]*)

В шаблон добавляется еще одно подвыражение: ([a-z A-Z]*). Оно почти совпадает с первым, но не содержит запятой. Последний элемент списка не завершается запятой.

○ ([a-z A-Z]*,)+([a-z A-Z]*){1}

Мы добавляем квантификатор {1}, чтобы разрешить вхождение ровно одного элемента списка без завершающей запятой.

○ ^([a-z A-Z]*,)+([a-z A-Z]*){1}\$

Наконец, метасимволы ^ и \$ привязывают потенциальное совпадение к началу и концу целевой строки. Это означает, что совпадением шаблона может быть только вся строка вместо некоторого подмножества ее символов.

Функция `REGEXP_LIKE` анализирует список имен и проверяет, соответствует ли он шаблону. Эта функция оптимизирована для простого обнаружения совпадения шаблона в строке, но другие функции способны на большее!

Поиск совпадения

Функция `REGEXP_INSTR` используется для поиска совпадений шаблона в строке. Общий синтаксис `REGEXP_INSTR`:

`REGEXP_INSTR (исходная_строка, шаблон [, начальная_позиция [, номер [, флаг_возвращаемого_значения`
`[, модификаторы [, подвыражение]]]])`

Здесь *исходная_строка* — строка, в которой выполняется поиск; *шаблон* — регулярное выражение, совпадение которого ищется в *исходной строке*; *начальная_позиция* — позиция, с которой начинается поиск; *номер* — порядковый номер совпадения (1 = первое, 2 = второе и т. д.); *флаг_возвращаемого_значения* — 0 для начальной позиции или 1 для конечной позиции совпадения; *модификаторы* — один или несколько модификаторов, управляющих процессом поиска (например, `i` для поиска без учета регистра). Начиная с Oracle11g, также можно задать параметр подвыражение (1 = первое, 2 = второе и т. д.), чтобы функция `REGEXP_INST` возвращала начальную позицию заданного подвыражения (части шаблона, заключенной в круглые скобки).

Например, чтобы найти первое вхождение имени, начинающегося с буквы А и завершающегося согласной буквой, можно использовать следующее выражение:

```
DECLARE
names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
names_adjusted VARCHAR2(61);
comma_delimited BOOLEAN;
j_location NUMBER;
BEGIN
-- Поиск по шаблону
comma_delimited := REGEXP_LIKE(names, '^([a-z ]*,)+([a-z ]*)$', 'i');

-- Продолжить, только если действительно был обнаружен список,
-- разделенный запятыми.
IF comma_delimited THEN
j_location := REGEXP_INSTR(names, 'A[a-z]*[^aeiou],|A[a-z]*[^aeiou]$');
DBMS_OUTPUT.PUT_LINE( j_location);
END IF;
END;
```

При выполнении этого фрагмента выясняется, что имя на букву А, завершающееся согласной (Andrew), начинается в позиции 22. А вот как проходило построение шаблона:

○ А

Совпадение начинается с буквы А. Беспокоиться о запятых не нужно — на этой стадии мы уже знаем, что работаем со списком, разделенным запятыми.

○ A[a-z]*

За буквой А следует некоторое количество букв или пробелов. Квантификатор `*` указывает, что за буквой А следует ноль или более таких символов.

○ A[a-z]*[^aeiou]

В выражение включается компонент `[^aeiou]`, чтобы имя могло заканчиваться любым символом, кроме гласной. Знак `^` инвертирует содержимое квадратных скобок — совпадает любой символ, кроме гласной буквы. Так как квантификатор не указан, требуется присутствие ровно одного такого символа.

○ A[a-z]*[^aeiou],

Совпадение должно завершаться запятой; в противном случае шаблон найдет совпадение для подстроки «An» в имени «Anna». Хотя добавление запятой решает эту

проблему, тут же возникает другая: шаблон не найдет совпадение для имени «Aagon» в конце строки.

○ `A[a-z]*[^aeiou], |A[a-z]*[^aeiou]$`

В выражении появляется вертикальная черта (`|`), обозначение альтернативы: общее совпадение находится при совпадении любого из вариантов. Первый вариант завершается запятой, второй — нет. Второй вариант учитывает возможность того, что текущее имя стоит на последнем месте в списке, поэтому он привязывается к концу строки метасимволом `$`.



Регулярные выражения — далеко не простая тема! Новички часто сталкиваются с нюансами обработки регулярных выражений, которые часто преподносят неприятные сюрпризы. Я потратил некоторое время на работу над этим примером и несколько раз зашел в тупик, прежде чем выйти на правильный путь. Не отчаивайтесь — с опытом писать регулярные выражения становится проще.

Функция `REGEXP_INSTR` приносит пользу в некоторых ситуациях, но обычно нас больше интересует текст совпадения, а не информация о его положении в строке.

Получение текста совпадения

Для демонстрации получения текста совпадения мы воспользуемся другим примером. Телефонные номера строятся по определенной схеме, но у этой схемы существует несколько разновидностей. Как правило, телефонный номер начинается с кода города (три цифры), за которым следует код станции (три цифры) и локальный номер (четыре цифры). Таким образом, телефонный номер представляет собой строку из 10 цифр. Однако существует много альтернативных способов представления числа. Код города может быть заключен в круглые скобки и обычно (хотя и не всегда) отделяется от остального номера пробелом, точкой или дефисом. Код станции обычно (но тоже не всегда) отделяется от локального номера пробелом, точкой или дефисом. Таким образом, любая из следующих форм записи телефонного номера является допустимой:

```
7735555253
773-555-5253
(773)555-5253
(773) 555 5253
773.555.5253
```

С подобными жесткими схемами записи легко работать при помощи регулярных выражений, но очень трудно без них. Мы воспользуемся функцией `REGEXP_SUBSTR` для извлечения телефонного номера из строки с контактными данными:

```
DECLARE
    contact_info VARCHAR2(200) := '
        address:
        1060 W. Addison St.
        Chicago, IL 60613
        home 773-555-5253
    ';
    phone_pattern VARCHAR2(90) :=
        '\(?:\d{3}\)?[[:space:]]\.\-]? \d{3}[[:space:]]\.\-]? \d{4}';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Номер телефона: ' ||
        REGEXP_SUBSTR(contact_info, phone_pattern, 1, 1));
END;
```

Результат работы программы:

```
Номер телефона: 773-555-5253
```

Ого! Шаблон получился довольно устрашающим. Давайте разобьем его на составляющие:

○ \(?

Шаблон начинается с необязательного символа открывающей круглой скобки. Так как круглые скобки в языке регулярных выражений являются *метасимволами* (то есть имеют специальное значение), для использования в качестве литерала символ круглой скобки необходимо *экранировать* (escape), поставив перед ним символ \ (обратная косая черта). Вопросительный знак — квантификатор, обозначающий ноль или одно вхождение предшествующего символа. Таким образом, эта часть выражения описывает необязательный символ открывающей круглой скобки.

○ \d{3}

\d — один из операторов, появившихся в Oracle10g Release 2 под влиянием языка Perl. Он обозначает произвольную цифру. Квантификатор {} указывает, что предшествующий символ входит в шаблон заданное количество раз (в данном случае три). Эта часть шаблона описывает три цифры.

○ \)?

Необязательный символ закрывающей круглой скобки.

○ [[:space:]\. \-]?

В квадратных скобках перечисляются символы, для которых обнаруживается совпадение — в данном случае это пропуск, точка или дефис. Конструкция [[:space:]] обозначает символьный класс POSIX для пропускных символов (пробел, табуляция, новая строка) в текущем наборе NLS. Точка и дефис являются метасимволами, поэтому в шаблоне их необходимо экранировать обратной косой чертой. Наконец, ? означает ноль или одно вхождение предшествующего символа. Эта часть шаблона описывает необязательный пропуск, точку или дефис.

○ \d{3}

Эта часть шаблона описывает три цифры (см. выше).

○ [[:space:]\. \-]?

Эта часть шаблона описывает необязательный пропуск, точку или дефис (см. выше).

○ \d{4}

Четыре цифры (см. выше).

Обязательно комментируйте свой код, использующий регулярные выражения, — это пригодится тому, кто будет разбираться в нем (вполне возможно, это будете вы сами через полгода).

Общий синтаксис REGEXP_SUBSTR:

REGEXP_SUBSTR (*исходная_строка*, *шаблон* [, *начальная_позиция* [, *номер* [, *модификаторы* [, *подвыражение*]]]])

Функция REGEXP_SUBSTR возвращает часть *исходной строки*, совпадающую с *шаблоном* или *подвыражением*. Если совпадение не обнаружено, функция возвращает NULL. Здесь *исходная_строка* — строка, в которой выполняется поиск; *шаблон* — регулярное выражение, совпадение которого ищется в *исходной строке*; *начальная_позиция* — позиция, с которой начинается поиск; *номер* — порядковый номер совпадения (1 = первое, 2 = второе и т. д.); *модификаторы* — один или несколько модификаторов, управляющих процессом поиска.

Начиная с Oracle11g, также можно задать параметр *подвыражение* (1 = первое, 2 = второе и т. д.), чтобы функция возвращала начальную позицию заданного подвыражения (части шаблона, заключенной в круглые скобки). Подвыражения удобны в тех случаях, когда требуется найти совпадение для всего шаблона, но получить совпадение только для его части. Скажем, если мы хотим найти телефонный номер и извлечь из него код

города, мы заключаем часть шаблона, описывающую код города, в круглые скобки, превращая ее в подвыражение:

```
DECLARE
  contact_info VARCHAR2(200) := '
    address:
    1060 W. Addison St.
    Chicago, IL 60613
    home 773-555-5253
    work (312) 555-1234
    cell 224.555.2233
  ';
  phone_pattern VARCHAR2(90) :=
    '\(?(d{3})\)?[[:space:]]\.\.-]?d{3}[[:space:]]\.\.-]?d{4}';
  contains_phone_nbr BOOLEAN;
  phone_number VARCHAR2(15);
  phone_counter NUMBER;
  area_code VARCHAR2(3);
BEGIN
  contains_phone_nbr := REGEXP_LIKE(contact_info,phone_pattern);
  IF contains_phone_nbr THEN
    phone_counter := 1;
    DBMS_OUTPUT.PUT_LINE('Номера:');
    LOOP
      phone_number := REGEXP_SUBSTR(contact_info,phone_pattern,1,phone_counter);
      EXIT WHEN phone_number IS NULL; -- NULL означает отсутствие совпадений
      DBMS_OUTPUT.PUT_LINE(phone_number);
      phone_counter := phone_counter + 1;
    END LOOP;
    phone_counter := 1;
    DBMS_OUTPUT.PUT_LINE('Коды городов:');
    LOOP
      area_code := REGEXP_SUBSTR(contact_info,phone_pattern,1,phone_
counter,'i',1);
      EXIT WHEN area_code IS NULL;
      DBMS_OUTPUT.PUT_LINE(area_code);
      phone_counter := phone_counter + 1;
    END LOOP;
  END IF;
END;
```

Этот фрагмент выводит телефонные номера и коды городов:

```
Номера:
773-555-5253
(312) 555-1234
224.555.2233
Коды городов:
773
312
224
```

Подсчет совпадений

Еще одна типичная задача — подсчет количества совпадений регулярного выражения в строке. До выхода Oracle11g программисту приходилось в цикле перебирать и подсчитывать совпадения. Теперь для этого можно воспользоваться новой функцией REGEXP_COUNT. Общий синтаксис ее вызова:

```
REGEXP_COUNT (исходная_строка, шаблон [,начальная_позиция [,модификаторы ]])
```

Здесь *исходная_строка* — строка, в которой выполняется поиск; *шаблон* — регулярное выражение, совпадение которого ищется в исходной строке; *начальная_позиция* — позиция, с которой начинается поиск; *модификаторы* — один или несколько модификаторов, управляющих процессом поиска.


```

DECLARE
  contact_info VARCHAR2(200) := '
    address:
    1060 W. Addison St.
    Chicago, IL 60613
    home 773-555-5253
    work (312) 123-4567';
  phone_pattern VARCHAR2(90) :=
    '\(?\d{3})\)?[[:space:]]\.\-]?(\d{3})[[:space:]]\.\-]?(\d{4})';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Обнаружено '
    || REGEXP_COUNT(contact_info, phone_pattern)
    || ' телефонных номера');
END;
```

Результат:

Обнаружено 2 телефонных номера

Замена текста

Поиск и замена — одна из лучших областей применения регулярных выражений. Текст замены может включать ссылки на части исходного выражения (называемые *обратными ссылками*), открывающие чрезвычайно мощные возможности при работе с текстом. Допустим, имеется список имен, разделенный запятыми, и его содержимое необходимо вывести по два имени в строке. Одно из решений заключается в том, чтобы заменить каждую вторую запятую символом новой строки. Сделать это при помощи стандартной функции REPLACE нелегко, но с функцией REGEXP_REPLACE задача решается просто. Общий синтаксис ее вызова:

```

REGEXP_REPLACE (исходная_строка, шаблон [, строка_замены
  [, начальная_позиция [, номер [, модификаторы ]]])
```

Здесь *исходная_строка* — строка, в которой выполняется поиск; *шаблон* — регулярное выражение, совпадение которого ищется в исходной строке; *начальная_позиция* — позиция, с которой начинается поиск; *модификаторы* — один или несколько модификаторов, управляющих процессом поиска. Пример:

```

DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
  names_adjusted VARCHAR2(61);
  comma_delimited BOOLEAN;
  extracted_name VARCHAR2(60);
  name_counter NUMBER;
BEGIN
  -- Искать совпадение шаблона
  comma_delimited := REGEXP_LIKE(names, '^[a-z ]*,)+([a-z ]*){1}$', 'i');
  -- Продолжать, только если мы действительно
  -- работаем со списком, разделенным запятыми.
  IF comma_delimited THEN
    names := REGEXP_REPLACE(
      names,
      '([a-z A-Z]*),([a-z A-Z]*)', '
        \1,\2' || chr(10) );
  END IF;
  DBMS_OUTPUT.PUT_LINE(names);
END;
```

Результат выглядит так:

```

Anna,Matt
Joe,Nathan
Andrew,Jeff
Aaron
```

При вызове функции `REGEXP_REPLACE` передаются три аргумента:

- `names` — исходная строка;
- `'([a-z A-Z]*)', '([a-z A-Z]*)', '` — выражение, описывающее заменяемый текст (см. ниже);
- `'\1, \2' || chr(10)` — текст замены. `\1` и `\2` — обратные ссылки, заложенные в основу нашего решения. Подробные объяснения также приводятся ниже.

Выражение, описывающее искомый текст, состоит из двух подвыражений в круглых скобках и двух запятых.

- `([a-z A-Z]*)`

Совпадение должно начинаться с имени.

- `,`

За именем должна следовать запятая.

- `([a-z A-Z]*)`

Затем идет другое имя.

- `,`

И снова одна запятая.

Наша цель — заменить каждую вторую запятую символом новой строки. Вот почему выражение написано так, чтобы оно совпадало с двумя именами и двумя запятыми. Также запятые не напрасно выведены за пределы подвыражений.

Первое совпадение для нашего выражения, которое будет найдено при вызове `REGEXP_REPLACE`, выглядит так:

`Anna,Matt,`

Два подвыражения соответствуют именам «Anna» и «Matt». В основе нашего решения лежит возможность сослаться на текст, совпавший с заданным подвыражением, через обратную ссылку. Обратные ссылки `\1` и `\2` в тексте замены ссылаются на текст, совпавший с первым и вторым подвыражением. Вот что происходит:

```
'\1,\2' || chr(10)      -- Текст замены
'Anna,\2' || chr(10)    -- Подстановка текста, совпавшего
                        -- с первым подвыражением
'Anna,Matt' || chr(10)  -- Подстановка текста, совпавшего
                        -- со вторым подвыражением
```

Вероятно, вы уже видите, какие мощные инструменты оказались в вашем распоряжении. Запятые из исходного текста попросту не используются. Мы берем текст, совпавший с двумя подвыражениями (имена «Anna» и «Matt»), и вставляем их в новую строку с одной запятой и одним символом новой строки.

Но и это еще не все! Текст замены легко изменить так, чтобы вместо запятой в нем использовался символ табуляции (ASCII-код 9):

```
names := REGEXP_REPLACE(
    names,
    '([a-z A-Z]*),([a-z A-Z]*),',
    '\1' || chr(9) || '\2' || chr(10) );
```

Теперь результаты выводятся в два аккуратных столбца:

```
Anna    Matt
Joe     Nathan
Andrew  Jeff
Aaron
```

Поиск и замена с использованием регулярных выражений — замечательная штука. Это мощный и элегантный механизм, с помощью которого можно сделать очень многое.

Максимализм и минимализм

Концепции максимализма и минимализма играют важную роль при написании регулярных выражений. Допустим, из разделенного запятыми списка имен нужно извлечь только первое имя и следующую за ним запятую. Список, уже приводившийся ранее, выглядит так:

```
names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
```

Казалось бы, нужно искать серию символов, завершающуюся запятой:

```
.*,
```

Давайте посмотрим, что из этого получится:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
BEGIN
  DBMS_OUTPUT.PUT_LINE( REGEXP_SUBSTR(names, '.*',')  );
END;
```

Результат выглядит так:

```
Anna,Matt,Joe,Nathan,Andrew,Jeff,
```

Совсем не то. Что произошло? Дело в «жадности» регулярных выражений: для каждого элемента регулярного выражения подыскивается *максимальное* совпадение, состоящее из как можно большего количества символов. Когда мы с вами видим конструкцию:

```
.*,
```

у нас появляется естественное желание остановиться у первой запятой и вернуть строку «Anna,». Однако база данных пытается найти самую длинную серию символов, завершающуюся запятой; база данных останавливается не на первой запятой, а на *последней*.

В версии Oracle Database 10g Release 1, в которой впервые была представлена поддержка регулярных выражений, возможности решения проблем максимализма были весьма ограничены. Иногда проблему удавалось решить изменением формулировки регулярного выражения — например, для выделения первого имени с завершающей запятой можно использовать выражение `[^,]*,`. Однако в других ситуациях приходилось менять весь подход к решению, часто вплоть до применения совершенно других функций.

Начиная с Oracle Database 10g Release 2, проблема максимализма отчасти упростилась с введением минимальных квантификаторов (по образцу тех, которые поддерживаются в Perl). Добавляя вопросительный знак к квантификатору после точки, то есть превращая `*` в `.*?`, я ищу самую короткую последовательность символов перед запятой:

```
DECLARE
  names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
BEGIN
  DBMS_OUTPUT.PUT_LINE( REGEXP_SUBSTR(names, '(.*)',')  );
END;
```

Теперь результат выглядит так, как и ожидалось:

```
Anna,
```

Минимальные квантификаторы останавливаются *на первом* подходящем совпадении, не пытаясь захватить *как можно больше* символов.

Подробнее о регулярных выражениях

Регулярные выражения на первый взгляд просты, но эта область на удивление глубока и нетривиальна. Они достаточно просты, чтобы вы начали пользоваться ими после прочтения этой главы (хочется надеяться!), и все же вам предстоит очень много узнать. Некоторые источники информации от компании Oracle и издательства O'Reilly:

- *Oracle Database Application Developer's Guide — Fundamentals*. В главе 4 этого руководства описана поддержка регулярных выражений в Oracle.
- *Oracle Regular Expression Pocket Reference*. Хороший вводный учебник по работе с регулярными выражениями; авторы — Джонатан Дженник (Jonathan Gennick) и Питер Линсли (Peter Linsley). Питер является одним из разработчиков реализации регулярных выражений Oracle.
- *Mastering Oracle SQL*. Одна из глав книги посвящена регулярным выражениям в контексте Oracle SQL. Отличный учебник для всех, кто хочет поднять свои навыки использования SQL на новый уровень.
- *Mastering Regular Expressions*. Книга Джеффри Фридла (Jeffrey Friedl) считается самым авторитетным источником информации об использовании регулярных выражений. Если вы хотите действительно глубоко изучить материал — читайте книгу Фридла.

Наконец, в приложении А приведена таблица с описаниями всех метасимволов, поддерживаемых реализацией регулярных выражений Oracle.

Работа с пустыми строками

Одна из причин постоянных недоразумений — особенно для тех программистов, которые перешли к Oracle от других СУБД и языков программирования, — заключается в том, что Oracle интерпретирует пустые строки как значение NULL. Это противоречит стандарту ISO SQL, в котором проводятся четкие различия между пустой строкой и строковой переменной, имеющей значение NULL:

```
/* Файл в Сети: empty_is_null.sql */
DECLARE
    empty_varchar2 VARCHAR2(10) := '';
    empty_char CHAR(10) := '';
BEGIN
    IF empty_varchar2 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('empty_varchar2 is NULL');
    END IF;
    IF '' IS NULL THEN
        DBMS_OUTPUT.PUT_LINE(''' is NULL');
    END IF;
    IF empty_char IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('empty_char is NULL');
    ELSEIF empty_char IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('empty_char is NOT NULL');
    END IF;
END;
```

Результат:

```
empty_varchar2 is NULL
'' is NULL
empty_char is NOT NULL
```

Данный пример показывает, что переменная типа CHAR не интерпретируется как NULL. Дело в том, что переменные CHAR как строки фиксированной длины в принципе не бывают пустыми. В приведенном примере она просто заполняется пробелами до длины 10 символов. А вот переменная VARCHAR2 равна NULL, поскольку ей присваивается строковый литерал нулевой длины.

На эту особенность строковых переменных следует обращать внимание при сравнении двух значений VARCHAR2 в командах IF. Помните, что значение NULL не может быть равно другому значению NULL. Возьмем программу, которая запрашивает у пользователя имя и сравнивает его со значением, прочитанным из базы данных:

```
DECLARE
  user_entered_name VARCHAR2(30);
  name_from_database VARCHAR2(30);
  ...
BEGIN
  ...
  IF user_entered_name <> name_from_database THEN
  ...
```

Если пользователь введет пустую строку вместо имени, результат проверки условия IF никогда не будет равным TRUE, поскольку значение NULL не бывает равным (или не равным) никакому другому значению. Альтернативный способ записи данного оператора IF выглядит так:

```
IF (user_entered_name <> name_from_database)
  OR (user_entered_name IS NULL) THEN
```

Это всего лишь один из способов решения проблемы равенства NULL пустой строки; он подходит не для каждой ситуации. В общем случае постарайтесь понять, чего вы пытаетесь добиться, учтите, что пустые строки будут интерпретироваться как NULL, и напишите соответствующий код.

Смешение значений CHAR и VARCHAR2

Если в коде PL/SQL используются строки как фиксированной (CHAR), так и переменной (VARCHAR2) длины, программист должен учитывать особенности выполнения в Oracle операций, в которых участвуют оба типа строк.

Перенос информации из базы данных в переменную

При выборке (SELECT или FETCH) данных из столбца базы данных типа CHAR и присваивании их переменной типа VARCHAR2 завершающие пробелы сохраняются. Если же перенести данные из столбца типа VARCHAR2 в переменную типа CHAR, PL/SQL автоматически дополнит извлекаемое значение пробелами до максимальной длины переменной. Таким образом, итоговое значение переменной определяется ее типом, а не типом столбца.

Перенос информации из переменной в базу данных

Когда вы помещаете (INSERT или UPDATE) значение переменной типа CHAR в столбец базы данных типа VARCHAR2, ядро SQL не подавляет завершающие пробелы. Например, при выполнении следующего кода PL/SQL столбцу `company_name` базы данных присваивается значение `ACME SHOWERS.....` (где символ «.» обозначает пробел). Иначе говоря, столбец дополняется пробелами до 20 символов, хотя исходным значением была строка из 12 символов.

```
DECLARE
  comp_id# NUMBER;
  comp_name CHAR(20) := 'ACME SHOWERS';
BEGIN
  SELECT company_id_seq.NEXTVAL
  INTO comp_id#
  FROM dual;
  INSERT INTO company (company_id, company_name)
  VALUES (comp_id#, comp_name);
END;
```

С другой стороны, при помещении значения переменной типа VARCHAR2 в столбец базы данных типа CHAR ядро SQL автоматически дополняет строку переменной длины пробелами до максимальной длины столбца (заданной при создании таблицы) и записывает результат в базу данных.

Сравнения строк

Предположим, некоторая программа содержит следующий оператор сравнения строк:

```
IF company_name = parent_company_name ...
```

PL/SQL должен сравнить значения `company_name` и `parent_company_name`. Эта операция выполняется одним из двух способов в зависимости от типов переменных:

- При сравнении двух переменных типа `CHAR` PL/SQL *дополняет их завершающими пробелами*.
- Если хотя бы одна из строк имеет переменную длину, PL/SQL выполняет сравнение *без дополнения завершающими пробелами*.

Различие между этими двумя методами сравнения продемонстрировано в следующем фрагменте кода:

```
DECLARE
    company_name CHAR(30)
        := 'Feuerstein and Friends';
    char_parent_company_name CHAR(35)
        := 'Feuerstein and Friends';
    varchar2_parent_company_name VARCHAR2(35)
        := 'Feuerstein and Friends';
BEGIN
    -- Сравняются два значения CHAR, строки дополняются пробелами
    IF company_name = char_parent_company_name THEN
        DBMS_OUTPUT.PUT_LINE ('Результат первой проверки равен TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Результат первой проверки равен FALSE');
    END IF;

    -- Сравняются CHAR и VARCHAR2, поэтому строки не дополняются пробелами
    IF company_name = varchar2_parent_company_name THEN
        DBMS_OUTPUT.PUT_LINE ('Результат второй проверки равен TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Результат второй проверки равен FALSE');
    END IF;
END;
```

Результат:

```
Результат первой проверки равен TRUE
Результат второй проверки равен FALSE
```

В первом сравнении участвуют два значения типа `CHAR`, поэтому используется дополнение завершающими пробелами: PL/SQL дополняет более короткое из двух значений до длины более длинного значения, после чего они сравниваются. В данном примере PL/SQL добавляет пять пробелов в конец значения переменной `company_name`, а затем сравнивает значения `company_name` и `char_parent_company_name`.

Проверка дает положительный результат. Обратите внимание: PL/SQL не изменяет значение переменной `company_name`. Это значение просто копируется в другую структуру памяти, где оно модифицируется для сравнения.

Во второй операции участвуют значения типов `CHAR` и `VARCHAR2`, поэтому PL/SQL выполняет сравнение без дополнения завершающими пробелами. Ни одно из значений не изменяется, а в сравнении используется их текущая длина. В данном случае первые 22 символа обеих строк одинаковы (Feuerstein and Friends), но значение переменной фиксированной длины `company_name` дополнено восемью пробелами, а значение переменной `varchar2_parent_company_name` — нет. В результате строки оказываются не равными.

Участие в проверке значения типа `VARCHAR2` приводит к тому, что сравнение выполняется без дополнения завершающими пробелами. Это верно и для выражений, содержащих более двух переменных, а также для выражений с оператором `IN`. Пример:

```
IF menu_selection NOT IN
    (save_and_close, cancel_and_exit, 'OPEN_SCREEN')
    THEN ...
```

Если хотя бы одна из четырех строк (`menu_selection`, две именованные константы и один литерал) объявлена как `VARCHAR2`, точное сравнение будет выполнено без модификации значений. Обратите внимание: строковые литералы (такие, как `OPEN_SCREEN`) всегда рассматриваются как строки фиксированной длины, то есть `CHAR`.

Символьные функции и аргументы типа CHAR

Символьная функция получает в качестве параметра одно или несколько символьных значений и возвращает символьное и числовое значение. Если символьная функция возвращает символьное значение, оно всегда имеет тип `VARCHAR2` (переменная длина) — кроме функций `UPPER` и `LOWER`. Эти функции преобразуют заданную строку к верхнему или нижнему регистру соответственно и возвращают значение фиксированной длины типа `CHAR`, если переданные в аргументах строки имели тип `CHAR`.

Краткая сводка строковых функций

Как упоминалось ранее, PL/SQL предоставляет в распоряжение программиста широкий набор мощных, высокоуровневых строковых функций для получения информации о строках и модификации их содержимого. Следующий список дает представление об их возможностях и синтаксисе. За полной информацией о конкретных функциях обращайтесь к справочнику Oracle SQL Reference.

○ ASCII(*символ*)

Возвращает числовой код (`NUMBER`) представления заданного символа в наборе символов базы данных.

○ ASCIISTR(*строка1*)

Получает строку в любом наборе символов и преобразует ее в строку ASCII-символов. Все символы, отсутствующие в кодировке ASCII, представляются в форме `\XXXX`, где `XXXX` — код символа в Юникоде.



За информацией о Юникоде и кодах символов обращайтесь на сайт <http://unicode.org>.

○ CHR(*код*)

Возвращает символ типа `VARCHAR2` (длина 1), соответствующий заданному коду. Функция является обратной по отношению к функции `ASCII`. У нее имеется разновидность, удобная при работе с данными в национальных наборах символов:

`CHR(код USING NCHAR_CS)`

Возвращает символ типа `NVARCHAR2` из национального набора символов.

○ COMPOSE(*строка1*)

Получает строку символов в формате Юникода и возвращает ее в нормализованном виде. Например, ненормализованное представление `'a\0303'` определяет символ `'a'` с тильдой сверху (то есть `ã`). Вызов `COMPOSE('a\0303')` возвращает значение `'\00E3'` — шестнадцатеричный код символа `ã` в Юникоде.



В Oracle9i Release 1 функция COMPOSE могла вызываться только из SQL-команд; в программах PL/SQL она использоваться не могла. Начиная с Oracle9i Release 2, функция COMPOSE также может использоваться в выражениях PL/SQL.

○ CONCAT(*строка1*, *строка2*)

Присоединяет *строку2* в конец *строки1*. Аналогичного результата можно добиться при помощи выражения *строка1* || *строка2*. Оператор || намного удобнее, поэтому функция CONCAT используется относительно редко.

○ CONVERT(*строка1*, *набор_символов*)

Преобразует строку из набора символов базы данных в заданный набор символов. При вызове также можно задать исходный набор символов:

CONVERT(*строка1*, *конечный_набор*, *исходный_набор*)

○ DECOMPOSE(*строка1*)

Получает строку в Юникоде и возвращает строку, в которой все составные символы разложены на элементы. Функция является обратной по отношению к COMPOSE. Например, вызов DECOMPOSE('ä') возвращает строку 'а~' (см. описание COMPOSE).

Существует две разновидности этой функции:

• DECOMPOSE(*строка1* CANONICAL)

Выполняет каноническую декомпозицию; полученный результат может быть восстановлен вызовом COMPOSE. Используется по умолчанию.

• DECOMPOSE(*строка1*)

Декомпозиция выполняется в так называемом режиме совместимости. Восстановление вызовом COMPOSE может оказаться невозможным.



Функция DECOMPOSE, как и COMPOSE, не может напрямую вызываться в выражениях PL/SQL в Oracle9i Release 1; ее необходимо вызывать из инструкций SQL. Начиная с Oracle9i Release 2, это ограничение было снято.

○ GREATEST(*строка1*, *строка2*, ...)

Получает одну или несколько строк и возвращает строку, которая оказалась бы последней (то есть наибольшей) при сортировке входных строк по возрастанию. Также см. описание функции LEAST, обратной по отношению к GREATEST.

○ INITCAP(*строка1*)

Изменяет регистр символов строкового аргумента, переводя первую букву каждого слова строки в верхний регистр, а остальные буквы — в нижний. Словом считается последовательность символов, отделенная от остальных символов пробелом или символом, не являющимся буквенно-цифровым (например, # или _). Например, вызов INITCAP('this is lower') дает результат 'This Is Lower'.

○ INSTR(*строка1*, *строка2*)

Возвращает позицию, с которой *строка2* входит в *строку1*; если вхождение не обнаружено, функция возвращает 0.

Существует несколько разновидностей этой функции:

• INSTR(*строка1*, *строка2*, *начальная_позиция*)

Поиск *строки2* в *строке1* начинается с позиции, заданной последним параметром. По умолчанию поиск начинается с позиции 1, так что вызов INSTR(string1, string2, 1) эквивалентен вызову INSTR(string1, string2).

- **INSTR(строка1, строка2, отрицательная_начальная_позиция)**
Смещение начальной позиции задается не от начала, а от конца *строки1*.
- **INSTR(строка1, строка2, начальная_позиция, n)**
Находит *n*-е вхождение *строки2*, начиная с заданной начальной позиции.
- **INSTR(строка1, строка2, отрицательная_начальная_позиция, n)**
Находит *n*-е вхождение *строки2*, начиная с заданной начальной позиции от конца *строки1*.

Функция **INSTR** рассматривает строку как последовательность символов. Ее разновидности **INSTRB**, **INSTR2** и **INSTR4** рассматривают строку как последовательность байтов, кодовых единиц (code units) или кодовых индексов (code points) Юникода соответственно. Разновидность **INSTRC** рассматривает строку как последовательность полных символов Юникода. Например, строка 'a\0303', которая представляет собой разложенный эквивалент '\00E3', или ã, рассматривается как один символ. Напротив, функция **INSTR** рассматривает 'a\0303' как последовательность из двух символов.

○ **LEAST(строка1, строка2, ...)**

Получает одну или несколько строк и возвращает строку, которая оказалась бы первой (то есть наименьшей) при сортировке входных строк по возрастанию. Также см. описание функции **GREATEST**, обратной по отношению к **LEAST**.

○ **LENGTH(строка1)**

Возвращает количество символов в строке. Разновидности **LENGTHB**, **LENGTH2** и **LENGTH4** возвращают количество байтов, кодовых единиц (code units) или кодовых индексов (code points) Юникода соответственно. Разновидность **LENGTHC** возвращает количество полных символов Юникода, нормализованных по мере возможности (то есть с преобразованием 'a\0303' в '\00E3').

Функция **LENGTH** обычно не возвращает нуль. Вспомните, что Oracle рассматривает пустую строку ('') как NULL, поэтому вызов **LENGTH('')** фактически эквивалентен попытке получить длину NULL; ее результат тоже будет равен NULL. Единственное исключение встречается при применении **LENGTH** к типу CLOB. Тип CLOB может содержать 0 байт и при этом быть отличным от NULL. В этом единственном случае **LENGTH** возвращает 0.

○ **LOWER(строка1)**

Преобразует все буквы заданной строки в нижний регистр. Функция является обратной по отношению к **UPPER**. Тип возвращаемого значения соответствует типу входных данных (CHAR, VARCHAR2, CLOB). Также см. NLS_LOWER.

○ **LPAD(строка1, итоговая_длина)**

Возвращает значение *строки1*, дополненное слева пробелами до *итоговой_длины*. У функции существует следующая разновидность:

- **LPAD(строка1, итоговая_длина, заполнитель)**

Присоединяет достаточное количество полных или частичных вхождений заполнителя, чтобы общая длина строки достигла заданной *итоговой_длины*. Например, вызов **LPAD('Merry Christmas!', 25, 'Ho! ')** вернет результат 'Ho! Ho! Ho! Merry Christmas!'.

У функции **LPAD** существует парная функция **RPAD**.

○ **LTRIM(строка1)**

Удаляет пробелы с левого края *строки1*. Также см. описания функций **TRIM** (стандарт ISO) и **RTRIM**. У функции существует следующая разновидность:

- **LTRIM(строка1, удаляемый_набор)**

Удаляет любые символы, входящие в строку *удаляемый_набор*, от левого края *строки1*.

○ NCHR(*код*)

Возвращает символ типа NVARCHAR2 (длина 1), соответствующий заданному коду. Функция CHR с условием USING NCHAR_CS реализует ту же функциональность, что и NCHR.

○ NLS_INITCAP(*строка1*)

Возвращает версию *строки1*, которая должна относиться к типу NVARCHAR2 или NCHAR, в которой первая буква каждого слова переводится в верхний регистр, а остальные буквы — в нижний. Функция возвращает значение типа VARCHAR2. «Словом» считается последовательность символов, отделенная от остальных символов пробелом или символом, не являющимся буквенно-цифровым.

Вы можете задать порядок сортировки, влияющий на определение «первой буквы»:

• NLS_INITCAP(*строка1*, 'NLS_SORT=*правило_сортировки*'))

В этой форме синтаксиса *правило_сортировки* представляет собой одно из допустимых названий правил сортировки, перечисленных в руководстве Oracle Database Globalization Support Guide, Appendix A, раздел «Linguistic Sorts».

Следующий пример показывает, чем функция INITCAP отличается от NLS_INITCAP:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(INITCAP('ijzer'));
  DBMS_OUTPUT.PUT_LINE(NLS_INITCAP('ijzer', 'NLS_SORT=XDUTCH'));
END;
```

Результат:

```
Ijzer
IJzer
```

В нидерландском языке последовательность символов «ij» рассматривается как один символ. Функция NLS_INITCAP распознает это обстоятельство при задании правила NLS_SORT и правильно преобразует символы слова «ijzer» («железо» по-нидерландски).

○ NLS_LOWER(*строка1*) и NLS_LOWER(*строка1*, 'NLS_SORT=*правило_сортировки*'))

Возвращает *строку1*, преобразованную в нижний регистр по правилам заданного языка. О том, как NLS_SORT может повлиять на результат преобразования, рассказано в описании функции NLS_INITCAP.

○ NLS_UPPER(*строка1*) и NLS_UPPER(*строка1*, 'NLS_SORT=*правило_сортировки*'))

Возвращает *строку1*, преобразованную в верхний регистр по правилам заданного языка. О том, как NLS_SORT может повлиять на результат преобразования, рассказано в описании функции NLS_INITCAP.

○ NLSSORT(*строка1*) и NLSSORT(*строка1*, 'NLS_SORT=*правило_сортировки*'))

Возвращает строку байтов, которая может использоваться для сортировки строкового значения по правилам заданного языка. Строка возвращается в формате RAW. Например, сравнение двух строк по правилам французского языка выполняется так:

```
IF NLSSORT(x, 'NLS_SORT=XFRENCH') > NLSSORT(y, 'NLS_SORT=XFRENCH') THEN...
```

Если второй параметр не указан, функция использует порядок сортировки по умолчанию, назначенный для сеанса. Полный список правил приведен в руководстве Oracle Database Globalization Support Guide, Appendix A, раздел «Linguistic Sorts».

○ REGEXP_COUNT, REGEXP_INSTR, REGEXP_LIKE, REGEXP_REPLACE, REGEXP_SUBSTR

За описаниями этих функций, предназначенных для работы с регулярными выражениями, обращайтесь к приложению A.

○ REPLACE(*строка1*, *искомая_строка*, *замена*)

Возвращает строку, полученную в результате замены всех вхождений *искомой_строки* в *строке1* строкой *замена*. Функция REPLACE может использоваться для замены всех вхождений определенной подстроки в одной инструкции.

○ REPLACE(*строка1*, *искомая_строка*)

Возвращает строку, полученную в результате удаления всех вхождений *искомой_строки* из *строки1*.

○ RPAD(*строка1*, *итоговая_длина*)

Возвращает значение *строки1*, дополненное справа пробелами до *итоговой_длины*. У функции существует следующая разновидность:

• RPAD(*строка1*, *итоговая_длина*, *заполнитель*)

Присоединяет достаточное количество полных или частичных вхождений *заполнителя*, чтобы общая длина строки достигла заданной *итоговой_длины*. Вызов RPAD('Merry Christmas!', 25, 'Ho! ') вернет результат 'Merry Christmas! Ho! Ho! '.

Функция RPAD дополняет строку справа, а парная ей функция LPAD — слева.

○ RTRIM(*строка1*)

Удаляет пробелы с правого края *строки1*. Также см. описания функций TRIM (стандарт ISO) и LTRIM. У функции существует следующая разновидность:

• RTRIM(*строка1*, *удаляемый_набор*)

Удаляет любые символы, входящие в строку *удаляемый_набор*, с правого края *строки1*.

○ SOUNDEX(*строка1*)

Возвращает строку с «фонетическим представлением» аргумента.

Пример:

```
SOUNDEX ('smith') --> 'S530'  
SOUNDEX ('SMYTHE') --> 'S530'  
SOUNDEX ('smith smith') --> 'S532'  
SOUNDEX ('smith z') --> 'S532'  
SOUNDEX ('feuerstein') --> 'F623'  
SOUNDEX ('feuerst') --> 'F623'
```

При использовании функции SOUNDEX следует помнить несколько правил:

- Значение SOUNDEX всегда начинается с первой буквы входной строки.
- Возвращаемое значение генерируется только по первым пяти согласным в строке.
- Для вычисления цифровой части SOUNDEX используются только согласные. Все гласные в строке, кроме начальных, игнорируются.
- Функция SOUNDEX игнорирует регистр символов; для букв верхнего и нижнего регистра генерируются одинаковые значения SOUNDEX.

Функция SOUNDEX полезна для запросов, при которых точное написание значения в базе данных неизвестно или не может быть легко определено.



Алгоритм SOUNDEX ориентирован на английский язык; в других языках он может работать плохо (или не работать вообще).

○ SUBSTR(*строка1*, *начальная_позиция*, *длина*)

Возвращает подстроку из *строки1*, которая начинается с *начальной_позиции* и имеет заданную длину. Если количество символов до конца *строки1* окажется меньше длины, возвращаются все символы от начальной позиции до конца строки. У функции существуют следующие разновидности:

• SUBSTR(*строка1*, *начальная_позиция*)

Возвращает все символы от *начальной_позиции* до конца *строки1*.

- `SUBSTR(строка1, отрицательная_начальная_позиция, длина)`
Начальная позиция подстроки отсчитывается от конца *строки1*.

- `SUBSTR(строка1, отрицательная_начальная_позиция)`
Возвращает последние `ABS(отрицательная_начальная_позиция)` строки.

Функция `SUBSTR` рассматривает строку как последовательность символов. Ее разновидности `SUBSTRB`, `SUBSTR2` и `SUBSTR4` рассматривают строку как последовательность байтов, кодовых единиц (code units) или кодовых индексов (code points) Юникода соответственно. Разновидность `SUBSTRC` рассматривает строку как последовательность полных символов Юникода. Например, строка `'a\0303'`, которая представляет собой разложенный эквивалент `'\00E3'`, или `ã`, рассматривается как один символ. Напротив, функция `SUBSTR` рассматривает `'a\0303'` как последовательность из двух символов.

- `TO_CHAR(национальные_символьные_данные)`

Преобразует данные из национального набора символов в эквивалентное представление в наборе символов базы данных. Также см. `TO_NCHAR`.



Функция `TO_CHAR` также может использоваться для преобразования даты/времени и чисел в удобочитаемую форму. Эти применения функции `TO_CHAR` описаны в главе 9 (для чисел) и главе 10 (для даты/времени).

- `TO_MULTI_BYTE(строка1)`

Преобразует однобайтовые символы в их многобайтовые эквиваленты. В некоторых многобайтовых кодировках, и прежде всего UTF-8, может существовать несколько вариантов представления одного символа. Скажем, в UTF-8 представление буквы 'б' может содержать от 1 до 4 байт. Для перехода от однобайтового представления к многобайтовому используется функция `TO_MULTI_BYTE`. Данная функция является обратной по отношению к `TO_SINGLE_BYTE`.

- `TO_NCHAR(символы_в_наборе_базы_данных)`

Преобразует данные из набора символов базы данных в эквивалентное представление в национальном наборе символов. Также см. `TO_CHAR` и `TRANSLATE...USING`.



Функция `TO_NCHAR` также может использоваться для преобразования даты/времени и чисел в удобочитаемую форму. Эти применения функции `TO_NCHAR` описаны в главе 9 (для чисел) и главе 10 (для даты/времени).

- `TO_SINGLE_BYTE(строка1)`

Преобразует многобайтовые символы в их однобайтовые эквиваленты. Функция является обратной по отношению к `TO_MULTI_BYTE`.

- `TRANSLATE(строка1, искомый_набор, набор_замены)`

Заменяет в *строке1* каждое вхождение символа из *искомого_набора* соответствующим символом *набора_замены*. Пример:

```
TRANSLATE ('abcd', 'ab', '12') --> '12cd'
```

Если *искомый_набор* содержит больше символов, чем *набор_замены*, «лишние» символы, не имеющие соответствия в *наборе_замены*, не включаются в результат. Пример:

```
TRANSLATE ('abcdefg', 'abcd', 'zyx') --> 'zyxefg'
```

Буква «d» удалена, потому что она присутствует в *искомом_наборе*, но не имеет эквивалента в *наборе_замены*. Функция `TRANSLATE` заменяет отдельные символы, а функция `REPLACE` — целые строки.

○ **TRANSLATE(текст USING CHAR_CS) и TRANSLATE(текст USING NCHAR_CS)**

Преобразует символьные данные в набор символов базы данных (**CHAR_CS**) или в национальный набор символов (**NCHAR_CS**). Выходным типом данных будет **VARCHAR2** или **NVARCHAR2** в зависимости от того, выполняется ли преобразование к набору символов базы данных или национальному набору символов соответственно.



Функция **TRANSLATE...USING** входит в число функций **SQL** по стандарту **ISO**. Начиная с **Oracle9i Release 1**, можно просто присвоить значение **VARCHAR2** переменной типа **NVARCHAR2**, и наоборот — система неявно выполнит нужное преобразование. Если же вы хотите выполнить преобразование явно, используйте функции **TO_CHAR** и **TO_NCHAR** для преобразования текста в набор символов базы данных и национальный набор символов соответственно. **Oracle** рекомендует пользоваться указанными функциями вместо **TRANSLATE...USING**, потому что они поддерживают более широкий набор входных типов данных.

○ **TRIM(FROM строка1)**

Возвращает строку, полученную в результате удаления из *строка1* всех начальных и конечных пробелов. У функции существуют следующие разновидности:

- **TRIM(LEADING FROM ...)**
Удаление только начальных пробелов.
- **TRIM(TRAILING FROM ...)**
Удаление только конечных пробелов.
- **TRIM(BOTH FROM ...)**
Удаление как начальных, так и конечных пробелов (используется по умолчанию).
- **TRIM(...удаляемый_символ FROM строка1)**
Удаление вхождений одного *удаляемого_символа* на выбор программиста.

Функция **TRIM** была включена в **Oracle8i** для обеспечения более полной совместимости со стандартом **ISO SQL**. Она сочетает в себе функциональность **LTRIM** и **RTRIM**, но отличается от них тем, что **TRIM** позволяет задать только один удаляемый символ, тогда как при использовании **LTRIM** и **RTRIM** можно задать набор удаляемых символов.

○ **UNISTR(строка1)**

Возвращает *строку1*, преобразованную в Юникод; таким образом, функция является обратной по отношению к **ASCIISTR**. Для представления непечатаемых символов во входной строке можно использовать запись **\XXXX**, где **XXXX** — кодовый индекс символа в Юникоде. Пример:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    UNISTR('Знак евро \20AC')
  );
END;
```

Знак евро €.

Функция предоставляет удобный доступ ко всему набору символов Юникода, в том числе и к тем, которые не могут вводиться непосредственно с клавиатуры. Юникод более подробно рассматривается в главе 25.

○ **UPPER(строка1)**

Преобразует все буквы заданной строки в верхний регистр. Тип возвращаемого значения соответствует типу входных данных (**CHAR**, **VARCHAR2**, **CLOB**). Функция является обратной по отношению к **LOWER**. Также см. **NLS_UPPER**.

9 Числа

И что бы мы делали без чисел? Хотя люди, которые не сильны в математике, предпочитают рассматривать любую информацию как текст, на практике большая часть информации в базах данных имеет числовую природу. Сколько единиц товара хранится на складе? Какую сумму мы задолжали? Насколько быстро развивается бизнес? Точные ответы на эти и многие другие вопросы можно получить именно в числовом выражении.

Для работы с числами в PL/SQL необходимо хотя бы в общих чертах изучить:

- числовые типы данных, имеющиеся в вашем распоряжении (и в каких ситуациях их уместно применять);
- преобразование числовых значений в текст и обратно;
- богатую библиотеку встроенных функций PL/SQL.

Все эти темы рассматриваются в настоящей главе. Начнем с типов данных.

Числовые типы данных

PL/SQL, как и PCУБД Oracle, поддерживает различные числовые типы данных для решения разных задач:

- **NUMBER** — тип с фиксированной точностью, идеально подходящий для работы с денежными суммами. Это единственный из числовых типов PL/SQL, реализация которого совершенно не зависит от платформы. Все операции с **NUMBER** должны работать одинаково независимо от оборудования, на котором работает программа.
- **PLS_INTEGER** и **BINARY_INTEGER** — целочисленные типы, соответствующие представлению целых чисел на вашем оборудовании. Арифметические операции выполняются на уровне машинных команд. Значения этих типов не могут храниться в базе данных.
- **SIMPLE_INTEGER** — тип появился в Oracle Database 11g. Используется для представления значений из того же диапазона, что и **BINARY_INTEGER**, но не допускает хранение **NULL** и не инициирует исключение при переполнении. Тип данных **SIMPLE_INTEGER** существенно ускоряет выполнение для кода, откомпилированного в машинный язык.
- **BINARY_FLOAT** и **BINARY_DOUBLE** — двоичные типы с плавающей запятой IEEE-754 одинарной и двойной точности. Я не рекомендую использовать эти типы для хранения денежных величин, но они могут пригодиться для быстрых вычислений с плавающей запятой.
- **SIMPLE_FLOAT** и **SIMPLE_DOUBLE** — типы появились в Oracle Database 11g. Они поддерживают тот же диапазон, что и **BINARY_FLOAT** с **BINARY_DOUBLE**, но не могут принимать

Возможно, вы также столкнетесь с другими числовыми типами — такими, как `FLOAT`, `INTEGER` и `DECIMAL`. Они представляют собой не что иное, как альтернативные имена для перечисленных мной основных числовых типов. Альтернативные имена более подробно рассматриваются в разделе «Числовые подтипы».

Безусловно, самый распространенный числовой тип данных в мире Oracle и PL/SQL. Используется для хранения целых чисел, а также чисел с фиксированной или плавающей запятой практически любого размера. До выхода Oracle10g тип **NUMBER** был единственным числовым типом, непосредственно поддерживавшимся ядром баз данных Oracle (в последующих версиях также поддерживаются **BINARY_FLOAT** и **BINARY_DOUBLE**). Тип **NUMBER** имеет платформенно-независимую реализацию, а вычисления с типом **NUMBER** всегда приводят к одинаковому результату независимо от того, на какой аппаратной платформе выполняется программа.

```
DECLARE
    x NUMBER;
```

[illegible][illegible]

При попытке явного присваивания переменной `NUMBER` слишком большого значения происходит исключение числового переполнения или потери значимости. Но в случае присваивания результата вычислений, превышающего самое большое допустимое значение, исключение не инициируется. Если приложению действительно необходимо работать с такими большими числами, придется либо организовать проверку диапазона, либо перейти на тип `BINARY_DOUBLE`, поддерживающий сравнение с `BINARY_DOUBLE_INFINITY`. С другой стороны, использование двоичных типов данных приводит к погрешностям округления; обязательно ознакомьтесь с описанием двоичных типов данных далее в этой главе. В большинстве приложений из-за ошибок округления предпочтение отдается типу `NUMBER`.

При объявлении переменной `NUMBER` можно задать для ее значения дополнительные параметры:

`NUMBER (A, B)`

Такое объявление определяет число с фиксированной запятой, где *A* — общее количество значащих цифр в числе, а *B* — количество цифр справа (положительное значение) или слева (отрицательное значение) от десятичной запятой. Оба параметра должны быть целочисленными литералами; ни переменные, ни константы в объявлении использоваться не могут. Допустимые значения параметра *A* находятся в диапазоне от 1 до 38, а параметра *B* — от -84 до 127.

При объявлении чисел с фиксированной запятой параметр *B* обычно меньше *A*. Например, переменную для хранения денежных сумм можно объявить как `NUMBER(9,2)`; это позволяет представлять значения до 9 999 999,99 включительно. Интерпретация этого объявления показана на рис. 9.1.

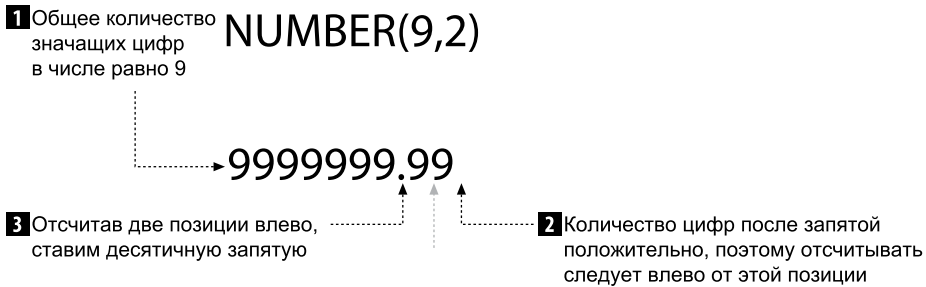


Рис. 9.1. Типичное объявление числа с фиксированной запятой

Как видно из рисунка, значение переменной `NUMBER(9,2)` представляет собой число с фиксированной запятой, состоящее из семи цифр слева от десятичной запятой и двух справа. Значения, хранимые в переменной, будут округляться максимум до сотых (табл. 9.1).

Попытка присваивания переменной двух последних значений вызывает исключение, поскольку для представления этих значений требуется больше цифр, чем помещается в переменной. Для хранения значений свыше 10 000 000 нужно минимум восемь значащих цифр в целой части числа. При округлении числа до семи цифр будут генерироваться ошибки. Ситуация становится более интересной при объявлении переменной, у которой количество цифр после десятичной запятой больше общего количества значащих цифр, или отрицательно. Пример представлен на рис. 9.2.

Переменная на рис. 9.2 содержит то же количество значащих цифр, что и переменная на рис. 9.1, но используются они по-другому. Поскольку параметр *B* равен 11, девять значащих цифр могут представлять только абсолютные значения меньше 0,01, которые

округляются до стомиллиардных. Результаты присваивания некоторых значений переменной типа NUMBER(9,11) приведены в табл. 9.2.

Таблица 9.1. Округление значений переменной NUMBER(9,2)

| Исходное значение | Округленное значение |
|-------------------|--|
| 1 234,56 | 1 234,56 |
| 1 234 567,984623 | 1 234 567,98 |
| 1 234 567,985623 | 1 234 567,99 |
| 1 234 567,995623 | 1 234 568,00 |
| 10 000 000,00 | Слишком большое значение — ошибка переполнения |
| −10 000 000,00 | То же |

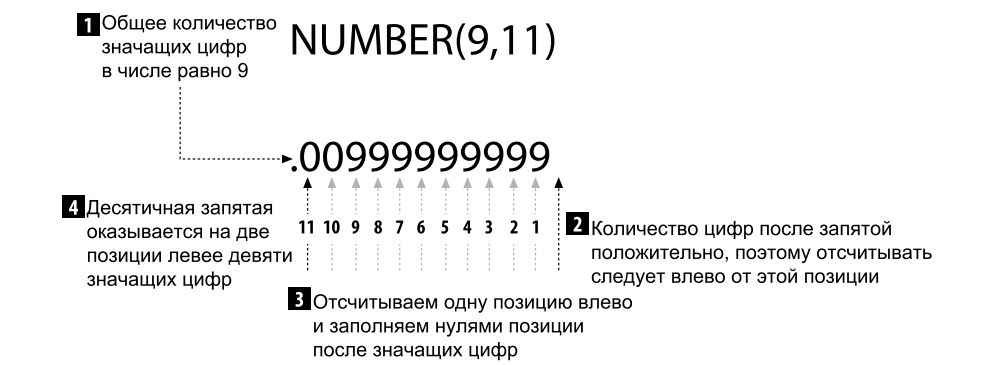


Рис. 9.2. Количество цифр после десятичной запятой больше общего количества значащих цифр

Таблица 9.2. Округление значений переменной NUMBER(9,11)

| Исходное значение | Округленное значение |
|-------------------|--|
| 0,00123456789 | 0,00123456789 |
| 0,0000000000005 | 0,000000000001 |
| 0,0000000000004 | 0,000000000000 |
| 0,01 | Слишком большое значение — ошибка переполнения |
| −0.01 | То же |

Если количество цифр в дробной части задано отрицательным значением, то десятичная запятая переносится вправо. Переменная, объявленная как NUMBER(9, -11), показана на рис. 9.3.

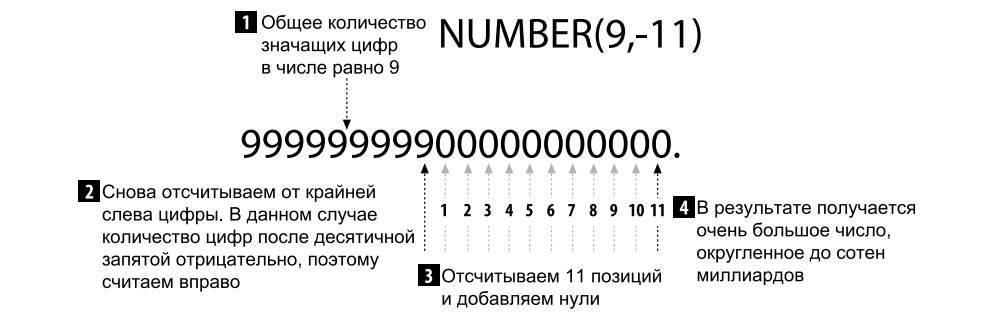


Рис. 9.3. Количество цифр после десятичной запятой задано отрицательным значением

а задавая параметры — переменные с фиксированной запятой. Если количество цифр после запятой указать равным нулю или не задавать вовсе, получится целочисленная переменная. Таким образом, один тип данных **NUMBER** покрывает все возможные варианты числовых значений.

Тип **PLS_INTEGER**

Тип данных **PLS_INTEGER** позволяет хранить целые числа в диапазоне от $-2\,147\,483\,648$ до $2\,147\,483\,647$. Значения хранятся в «родном» целочисленном формате аппаратной платформы. Несколько примеров объявлений переменных типа **PLS_INTEGER**:

```
DECLARE
    loop_counter PLS_INTEGER;
    days_in_standard_year CONSTANT PLS_INTEGER := 365;
    emp_vacation_days PLS_INTEGER DEFAULT 14;
```

Тип данных **PLS_INTEGER** был разработан для увеличения скорости вычислений. До выхода Oracle10g тип **PLS_INTEGER** был единственным целочисленным типом, использовавшим машинные вычисления. Все остальные типы данных использовали математическую библиотеку языка C. В результате операции со значениями типа **PLS_INTEGER** выполняются быстрее операций со значениями **NUMBER**. А поскольку значения **PLS_INTEGER** целочисленные, проблем совместимости при переходе с одной платформы на другую из-за них практически не бывает.

Используйте **PLS_INTEGER**, если ваша программа интенсивно выполняет целочисленные операции. Однако следует помнить, что в выражениях, где приходится выполнять частые преобразования к типу **NUMBER** и обратно, лучше изначально использовать тип **NUMBER**. Наибольшая эффективность достигается при использовании **PLS_INTEGER** для целочисленной арифметики (и счетчиков циклов) там, где удастся избежать многочисленных преобразований к типу **NUMBER** и обратно. Как показывает следующий пример, при использовании этого типа в целочисленной арифметике полученные значения округляются до целых чисел:

```
DECLARE
    int1 PLS_INTEGER;
    int2 PLS_INTEGER;
    int3 PLS_INTEGER;
    nbr NUMBER;
BEGIN
    int1 := 100;
    int2 := 49;
    int3 := int2/int1;
    nbr := int2/int1;
    DBMS_OUTPUT.PUT_LINE('integer 100/49 =' || TO_CHAR(int3));
    DBMS_OUTPUT.PUT_LINE('number 100/49 =' || TO_CHAR(nbr));
    int2 := 50;
    int3 := int2/int1;
    nbr := int2/int1;
    DBMS_OUTPUT.PUT_LINE('integer 100/50 =' || TO_CHAR(int3));
    DBMS_OUTPUT.PUT_LINE('number 100/50 =' || TO_CHAR(nbr));
END;
```

Программа выводит следующий результат:

```
integer 100/49 =0
number 100/49 =.49
integer 100/50 =1
number 100/50 =.5
```

Если итоговое значение целочисленной операции выходит за пределы диапазона допустимых значений (от $-2\,147\,483\,648$ до $2\,147\,483\,647$), произойдет ошибка целочисленного переполнения.

Тип `BINARY_INTEGER`

Тип данных `BINARY_INTEGER` позволяет хранить целые числа со знаком в двоичном формате. Семантика этого типа данных изменилась в Oracle10g Release 1. Начиная с этой версии тип `BINARY_INTEGER` стал эквивалентным `PLS_INTEGER`. В Oracle9i Release 2 и более ранних версиях тип `BINARY_INTEGER` отличался от `PLS_INTEGER` тем, что он был реализован с использованием платформенно-независимого библиотечного кода.

Любопытная подробность: казалось бы, в пакете `STANDARD` тип `BINARY_INTEGER` ограничивается значениями от $-2\,147\,483\,647$ до $2\,147\,483\,647$, однако в моей программе не инициировались исключения при присваивании значений из диапазона от $-2\,147\,483\,648$ до $2\,147\,483\,647$ (немного расширенного в отрицательной части):

```
subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
```

Тип `BINARY_INTEGER` не рекомендуется использовать в новых разработках — разве что вам потребуется, чтобы код работал в старых версиях Oracle до 7.3 (версия, в которой появился тип `PLS_INTEGER`). Надеюсь, вам не приходится иметь дела с такими древностями!

Тип `SIMPLE_INTEGER`

Тип `SIMPLE_INTEGER` появился в Oracle11g. Он представляет собой оптимизированную по быстродействию версию `PLS_INTEGER` с некоторыми ограничениями. Тип `SIMPLE_INTEGER` имеет такой же диапазон значений, как `PLS_INTEGER` (от $-2\,147\,483\,648$ до $2\,147\,483\,647$), но не поддерживает `NULL` и не проверяет условия переполнения. Казалось бы, зачем использовать этот неполноценный дубликат `PLS_INTEGER`? Если ваша программа компилируется в машинный код, а ситуация такова, что `NULL` и переполнение исключены, тип `SIMPLE_INTEGER` обеспечит значительно лучшее быстродействие. Рассмотрим следующий пример:

```
/* Файл в Сети: simple_integer_demo.sql */
-- Начнем с создания процедуры, требующей большого объема
-- вычислений, с использованием PLS_INTEGER
CREATE OR REPLACE PROCEDURE pls_test (iterations IN PLS_INTEGER)
AS
    int1      PLS_INTEGER := 1;
    int2      PLS_INTEGER := 2;
    begints   timestamp;
    endts     timestamp;
BEGIN
    begints := SYSTIMESTAMP;

    FOR cnt IN 1 .. iterations
    LOOP
        int1 := int1 + int2 * cnt;
    END LOOP;

    endts := SYSTIMESTAMP;
    DBMS_OUTPUT.put_line(
        || iterations
        || ' итераций за время:'
        || TO_CHAR (endts - begints));
END;
/

-- Затем та же процедура создается с использованием SIMPLE_INTEGER
CREATE OR REPLACE PROCEDURE simple_test (iterations IN SIMPLE_INTEGER)
AS
    int1      SIMPLE_INTEGER := 1;
    int2      SIMPLE_INTEGER := 2;
    begints   timestamp;
    endts     timestamp;
```

```

BEGIN
  begints := SYSTIMESTAMP;
  FOR cnt IN 1 .. iterations
  LOOP
    int1 := int1 + int2 * cnt;
  END LOOP;
  endts := SYSTIMESTAMP;
  DBMS_OUTPUT.put_line(  iterations
                        || ' итераций за время:'
                        || TO_CHAR (endts - begints));
END;
/

-- Сначала процедуры перекомпилируются в режиме интерпретации
ALTER PROCEDURE pls_test COMPILE PLSQL_CODE_TYPE=INTERPRETED;
/

ALTER PROCEDURE simple_test COMPILE PLSQL_CODE_TYPE=INTERPRETED
/

-- Сравнить время выполнения
BEGIN pls_test(123456789); END;
/
123456789 итераций за время:+000000000 00:00:06.375000000

BEGIN simple_test(123456789); END;
/
123456789 итераций за время:+000000000 00:00:06.000000000

-- Перекомпиляция в машинный код
ALTER PROCEDURE pls_test COMPILE PLSQL_CODE_TYPE=NATIVE
/

ALTER PROCEDURE simple_test COMPILE PLSQL_CODE_TYPE= NATIVE
/

-- Сравнение времени выполнения
BEGIN pls_test(123456789); END;
/
123456789 итераций за время:+000000000 00:00:03.703000000

BEGIN simple_test(123456789); END;
/
123456789 итераций за время:+000000000 00:00:01.203000000

```

Из этого примера видно, что тип `SIMPLE_INTEGER` обеспечивает небольшое преимущество по быстродействию интерпретируемого кода (6% в тесте на сервере Microsoft Windows). Оба типа, `PLS_INTEGER` и `SIMPLE_INTEGER` быстрее работают при компиляции в машинный код, но в «машинном» варианте `SIMPLE_INTEGER`, работает на 300% быстрее, чем `PLS_INTEGER`! В качестве упражнения проведите этот тест с типом `NUMBER` — я обнаружил, что `SIMPLE_INTEGER` превосходит `NUMBER` по производительности на 1000%. На сервере Linux с Oracle Database 11g Release 2 аналогичные различия в производительности наблюдались при использовании `SIMPLE_INTEGER` (часто на несколько сотен процентов быстрее, чем с альтернативными числовыми типами).

Типы `BINARY_FLOAT` и `BINARY_DOUBLE`

В Oracle10g появились два новых вещественных типа: `BINARY_FLOAT` и `BINARY_DOUBLE`. Они соответствуют вещественным типам с одинарной и двойной точностью, определенным в стандарте IEEE-754. Эти типы реализуются как PL/SQL, так и самим ядром базы данных, поэтому они могут использоваться и в определениях таблиц, и в коде PL/SQL. В табл. 9.4 эти новые типы сравниваются с популярным типом `NUMBER`.

Таблица 9.4. Сравнение вещественных типов

| Характеристика | BINARY_FLOAT | BINARY_DOUBLE | NUMBER |
|--|--------------------|-------------------------|---------------------------------|
| Максимальное абсолютное значение | 3,40282347E+38F | 1,7976931348623157E+308 | 9.999...999E+121 (38 «девяток») |
| Минимальное абсолютное значение | 1.17549435E-38F | 2.2250748585072014E-308 | 1.0E-127 |
| Количество байтов, используемое для значения | 4 (32 бита) | 8 (64 бита) | от 1 до 20 |
| Количество байтов длины | 0 | 0 | 1 |
| Представление | Двоичное, IEEE-754 | Двоичное, IEEE-754 | Десятичное |
| Суффикс литералов | f | d | Нет |

При записи литералов этих новых типов добавляется суффикс **f** или **d** — в зависимости от того, должен ли литерал интерпретироваться как **BINARY_FLOAT** или **BINARY_DOUBLE**.
Пример:

```
DECLARE
  my_binary_float  BINARY_FLOAT  := .95f;
  my_binary_double BINARY_DOUBLE := .95d;
  my_number        NUMBER        := .95;
```

Также существуют специальные литералы, используемые при работе с вещественными типами IEEE-754. Следующие литералы поддерживаются как в PL/SQL, так и в SQL:

- **BINARY_FLOAT_NAN** и **BINARY_DOUBLE_NAN** — «не число» одинарной или двойной точности соответственно.
- **BINARY_FLOAT_INFINITY** и **BINARY_DOUBLE_INFINITY** — бесконечность одинарной или двойной точности соответственно.

Другая группа литералов поддерживается *только* в PL/SQL:

- **BINARY_FLOAT_MIN_NORMAL**, **BINARY_FLOAT_MAX_NORMAL** — границы нормального диапазона значений для переменных с одинарной и двойной точностью соответственно.
- **BINARY_FLOAT_MIN_SUBNORMAL**, **BINARY_DOUBLE_MIN_SUBNORMAL** — границы *субнормального* диапазона значений — части стандарта IEEE-754, предназначенной для снижения риска, связанного с потерей значимости.

Наконец, при работе с этими типами данных используются следующие предикаты:

- **IS NAN** и **IS NOT NAN** — проверяет, является ли значение IEEE-754 «не числом».
- **IS INFINITE** и **IS NOT INFINITE** — проверяет, представляет ли значение IEEE-754 бесконечность.

Очень важно помнить, что типы **BINARY** являются двоичными. Их не рекомендуется использовать ни в каких ситуациях, требующих точного десятичного представления. Следующий пример показывает, почему эти типы не должны, например, использоваться для представления денежных сумм:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(0.95f); --BINARY_FLOAT
  DBMS_OUTPUT.PUT_LINE(0.95d); --BINARY_DOUBLE
  DBMS_OUTPUT.PUT_LINE(0.95);  --NUMBER
END;
```

Программа выводит следующий результат:

```
9.49999988E-001
9.4999999999999996E-001
.95
```

По аналогии с тем, как некоторые дроби (например, $1/3$) невозможно точно представить в виде десятичного числа, в некоторых случаях десятичные числа — как, например,

0,95 — не имеют точного представления в двоичном виде. При работе с денежными суммами следует использовать тип **NUMBER**.



Будьте внимательны при смешении вещественных типов при сравнениях, например:

```
BEGIN
  IF 0.95f = 0.95d
  THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
  IF ABS(0.95f - 0.95d) < 0.000001d
  THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END;
```

Программа выводит следующий результат:

```
FALSE
TRUE
```

Вывод FALSE и TRUE соответственно демонстрирует коварные проблемы, с которыми можно столкнуться при представлении десятичных значений в двоичной форме. Представление 0,95 в формате **BINARY_DOUBLE** содержит больше цифр, чем версия **BINARY_FLOAT**, поэтому эти два значения при сравнении не считаются равными. Второе сравнение дает результат TRUE; чтобы компенсировать невозможность точного представления 0,95 в двоичной форме, мы произвольно решаем, что два значения считаются равными, если разность между ними меньше одной миллионной.

Для чего используются типы IEEE-754? Первая причина — производительность, вторая — соответствие стандартам IEEE. При выполнении масштабных вычислений использование типов IEEE-754 может привести к заметному выигрышу по скорости. Следующий программный блок выводит время, необходимое для вычисления площади 5 000 000 кругов и вычисления 5 000 000 синусов. Обе задачи выполняются дважды — с **BINARY_DOUBLE** и **NUMBER**:

```
/* Файл в Сети: binary_performance.sql */
DECLARE
  bd BINARY_DOUBLE;
  bd_area BINARY_DOUBLE;
  bd_sine BINARY_DOUBLE;
  nm NUMBER;
  nm_area NUMBER;
  nm_sine NUMBER;
  pi_bd BINARY_DOUBLE := 3.1415926536d;
  pi_nm NUMBER := 3.1415926536;
  bd_begin TIMESTAMP(9);
  bd_end TIMESTAMP(9);
  bd_wall_time INTERVAL DAY TO SECOND(9);
  nm_begin TIMESTAMP(9);
  nm_end TIMESTAMP(9);
  nm_wall_time INTERVAL DAY TO SECOND(9);
BEGIN
  -- Площадь круга вычисляется 5 000 000 раз с BINARY_DOUBLE
  bd_begin := SYSTIMESTAMP;
  bd := 1d;
  LOOP
    bd_area := bd * bd * pi_bd;
    bd := bd + 1d;
    EXIT WHEN bd > 5000000;
  END LOOP;
```

```

bd_end := SYSTIMESTAMP;

-- Площадь круга вычисляется 5 000 000 раз с NUMBER
nm_begin := SYSTIMESTAMP;
nm := 1;
LOOP
    nm_area := nm * nm * 2 * pi_nm;
    nm := nm + 1;
    EXIT WHEN nm > 5000000;
END LOOP;
nm_end := SYSTIMESTAMP;

-- Вычисление и вывод затраченного времени
bd_wall_time := bd_end - bd_begin;
nm_wall_time := nm_end - nm_begin;
DBMS_OUTPUT.PUT_LINE('BINARY_DOUBLE area = ' || bd_wall_time);
DBMS_OUTPUT.PUT_LINE('NUMBER area = ' || nm_wall_time);

-- Синус вычисляется 5 000 000 раз с BINARY_DOUBLE
bd_begin := SYSTIMESTAMP;
bd := 1d;
LOOP
    bd_sine := sin(bd);
    bd := bd + 1d;
    EXIT WHEN bd > 5000000;
END LOOP;
bd_end := SYSTIMESTAMP;

-- Синус вычисляется 5 000 000 раз с NUMBER
nm_begin := SYSTIMESTAMP;
nm := 1;
LOOP
    nm_sine := sin(nm);
    nm := nm + 1;
    EXIT WHEN nm > 5000000;
END LOOP;
nm_end := SYSTIMESTAMP;

-- Вычисление и вывод затраченного времени
bd_wall_time := bd_end - bd_begin;
nm_wall_time := nm_end - nm_begin;
DBMS_OUTPUT.PUT_LINE('BINARY_DOUBLE sine = ' || bd_wall_time);
DBMS_OUTPUT.PUT_LINE('NUMBER sine = ' || nm_wall_time);
END;
```

Мои результаты, которые получились довольно стабильными для серии запусков, выглядели так:

```

BINARY_DOUBLE area = +00 00:00:02.792692000
NUMBER area = +00 00:00:08.942327000
BINARY_DOUBLE sine = +00 00:00:04.149930000
NUMBER sine = +00 00:07:37.596783000
```

Но не надо слишком доверять тестам — в том числе и тем, которые я только что привел! Как видно из примера, диапазон возможного прироста производительности от использования типа IEEE-754 вместо NUMBER весьма велик. При использовании BINARY_DOUBLE вычисление площади круга 5 миллионов раз занимает приблизительно 40% времени от его вычисления с использованием NUMBER. Однако при вычислении синуса 5 миллионов раз задача решается всего за 0,9% времени. В каждой конкретной ситуации выигрыш зависит от используемых вычислений. Из всего сказанного не стоит делать вывод, что типы IEEE-754 позволяют выполнить работу на фиксированные X% быстрее, чем NUMBER. Скорее речь идет о том, что потенциальный прирост производительности от использования типов IEEE-754 вместо NUMBER может быть заметным и его стоит принять во внимание при выполнении масштабных вычислений.

СМЕШАННОЕ ИСПОЛЬЗОВАНИЕ ТИПОВ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Oracle устанавливает систему приоритетов для неявных преобразований типов с плавающей запятой. В порядке убывания приоритетов типы образуют последовательность `BINARY_DOUBLE`, `BINARY_FLOAT` и `NUMBER`. Когда вы пишете выражение, в котором используется комбинация этих типов, база данных пытается преобразовать все значения к наивысшему приоритету из выражения. Например, если в выражении используются `BINARY_FLOAT` и `NUMBER`, Oracle сначала преобразует все значения к типу `BINARY_FLOAT`.

Если вы не хотите, чтобы база данных выполняла эти неявные преобразования, используйте функции `TO_NUMBER`, `TO_BINARY_FLOAT` и `TO_BINARY_DOUBLE`:

```
DECLARE
  nbr NUMBER := 0.95;
  bf BINARY_FLOAT := 2;
  nbr1 NUMBER;
  nbr2 NUMBER;
BEGIN
  -- Приоритет по умолчанию, повысить до binary_float
  nbr1 := nbr * bf;
  -- Понизить BINARY_FLOAT до NUMBER
  nbr2 := nbr * TO_NUMBER(bf);
  DBMS_OUTPUT.PUT_LINE(nbr1);
  DBMS_OUTPUT.PUT_LINE(nbr2);
END;
```

Результат выглядит так:

```
1.89999998
1.9
```

Чтобы избежать неоднозначности и возможных ошибок с неявными преобразованиями, я рекомендую применять явные преобразования — например, с применением функций `TO_NUMBER`, `TO_BINARY_FLOAT` и `TO_BINARY_DOUBLE`.

Впрочем, в некоторых областях реализация двоичных вещественных типов Oracle не полностью соответствует стандарту IEEE-754. Например, Oracle преобразует `-0` в `+0`, тогда как стандарт IEEE-754 такого поведения не требует. Если совместимость со стандартом важна для вашего приложения, обратитесь к разделу «Типы данных» руководства *SQL Reference* от Oracle — в нем содержится точная информация о том, где и как Oracle отклоняется от стандарта IEEE-754.

Типы `SIMPLE_FLOAT` и `SIMPLE_DOUBLE`

Типы данных `SIMPLE_FLOAT` и `SIMPLE_DOUBLE` появились в Oracle11g. Они представляют собой оптимизированные по быстродействию версии `BINARY_FLOAT` и `BINARY_DOUBLE` с некоторыми ограничениями. Типы `SIMPLE_FLOAT` и `SIMPLE_DOUBLE` имеют такие же диапазоны значений, как `BINARY_FLOAT` и `BINARY_DOUBLE`, но не поддерживают значения `NULL`, специальные литералы IEEE (`BINARY_FLOAT_NAN`, `BINARY_DOUBLE_INFINITY` и т. д.), специальные предикаты IEEE (`IS NAN`, `IS INFINITY` и т. д.). Кроме того, они не проверяют условие переполнения. Как и тип `SIMPLE_INTEGER`, эти специализированные типы в соответствующей ситуации способны значительно ускорить выполнение кода.

Числовые подтипы

Oracle также поддерживает ряд числовых подтипов данных. Большая их часть представляет собой альтернативные имена для трех описанных нами базовых типов данных.

Подтипы введены для достижения совместимости с типами данных ISO SQL, SQL/DS и DB2 и обычно имеют те же диапазоны допустимых значений, что и их базовые типы. Однако иногда значения подтипа ограничены некоторым подмножеством значений базового типа. Подтипы числовых данных представлены в табл. 9.5.

Таблица 9.5. Предопределенные числовые подтипы данных

| Подтип | Совместимость | Соответствующий тип данных Oracle |
|---------------------------|---------------|--|
| DEC (A, B) | ANSI | NUMBER (A,B) |
| DECIMAL (A, B) | IBM | NUMBER (A,B) |
| DOUBLE PRECISION | ANSI | NUMBER, точность 126 двоичных цифр |
| FLOAT | ANSI, IBM | NUMBER, точность 126 двоичных цифр |
| FLOAT (двоичная_точность) | ANSI, IBM | NUMBER, с точностью до 126 двоичных цифр (по умолчанию) |
| INT | ANSI | NUMBER(38) |
| INTEGER | ANSI, IBM | NUMBER(38) |
| NATURAL | N/A | PLS_INTEGER*, но только с неотрицательными значениями (0 и выше) |
| NATURALN | N/A | То же, что NATURAL, но с запретом NULL |
| NUMERIC (A, B) | ANSI | NUMBER (A,B) |
| POSITIVE | N/A | PLS_INTEGER*, но только с положительными значениями (1 и выше) |
| POSITIVEN | N/A | То же, что POSITIVE, но с запретом NULL |
| REAL | ANSI | NUMBER, точность 63 цифры |
| SIGNTYPE | N/A | PLS_INTEGER* с возможными значениями -1, 0 и 1 |
| SMALLINT | ANSI, IBM | NUMBER (38) |

* *BINARY_INTEGER* до Oracle10g.

Типы данных NUMERIC, DECIMAL и DEC позволяют объявлять только значения с фиксированной запятой. Типы DOUBLE PRECISION и REAL эквивалентны NUMBER. С помощью типа FLOAT можно объявлять числа с плавающей запятой с двойной точностью в диапазоне от 63 до 126 бит. Возможность определения точности числа в битах, а не в цифрах неудобна, и, скорее всего, вам не придется использовать типы данных ISO/IBM.

На практике часто используются подтипы PLS_INTEGER, к которым относятся NATURAL и POSITIVE. Они ограничивают значения, которые могут храниться в переменной, а их применение делает логику программы более понятной. Например, если переменная может принимать только неотрицательные значения, ее можно объявить с типом NATURAL (0 и выше) или POSITIVE (1 и выше). Такое объявление будет способствовать самодocumentированию кода.

Числовые преобразования

Компьютеры лучше работают с числами в двоичном представлении, тогда как людям удобнее видеть числовые данные в виде строк, состоящих из цифр, запятых и пробелов. PL/SQL позволяет преобразовывать числа в строки, и наоборот. Обычно такие преобразования выполняются функциями TO_CHAR и TO_NUMBER.



При работе с двоичными вещественными типами IEEE-754 используйте функции TO_BINARY_FLOAT и TO_BINARY_DOUBLE. Чтобы упростить последующее изложение, в тексте будет упоминаться только функция TO_NUMBER. Помните, что все неуточненные ссылки на TO_NUMBER в равной степени относятся к функциям TO_BINARY_FLOAT и TO_BINARY_DOUBLE.

Функция TO_NUMBER

Функция TO_NUMBER преобразует строки фиксированной и переменной длины, а также вещественные типы IEEE-754 к типу NUMBER с использованием необязательной маски форматирования. Используйте эту функцию, когда потребуется преобразовать строковое представление числа в соответствующее числовое значение. Синтаксис вызова TO_NUMBER:

TO_NUMBER(*строка* [, *формат* [, *параметры_nls*]])

Здесь *строка* — строка или выражение типа BINARY_DOUBLE, содержащее представление числа; *формат* — необязательная маска, которая определяет, как функция TO_NUMBER должна интерпретировать символьное представление числа, содержащегося в первом параметре; *параметры_nls* — необязательная строка со значениями параметров NLS. Ее можно применять для замены текущих установок параметров NLS уровня сеанса.



При работе с функциями TO_BINARY_FLOAT и TO_BINARY_DOUBLE можно использовать строки INF и -INF для представления положительной и отрицательной бесконечности, а также строку NaN для представления «не числа». Эти специальные строки не чувствительны к регистру символов.

Использование TO_NUMBER без параметров форматирования

В простейших случаях функция TO_NUMBER вызывается без строки форматирования. Все следующие преобразования успешно выполняются без дополнительных параметров:

```
DECLARE
  a NUMBER;
  b NUMBER;
  c NUMBER;
  d NUMBER;
  e BINARY_FLOAT;
  f BINARY_DOUBLE;
  g BINARY_DOUBLE;

  n1 VARCHAR2(20) := '-123456.78';
  n2 VARCHAR2(20) := '+123456.78';
BEGIN
  a := TO_NUMBER('123.45');
  b := TO_NUMBER(n1);
  c := TO_NUMBER(n2);
  d := TO_NUMBER('1.25E2');
  e := TO_BINARY_FLOAT('123.45');
  f := TO_BINARY_DOUBLE('inf');
  g := TO_BINARY_DOUBLE('NaN');
END;
```

В общем случае функция TO_NUMBER может использоваться без параметров форматирования в следующих случаях:

- когда число представлено только цифрами с единственной десятичной запятой;
- при использовании научной записи — например, 1.25E2;
- перед числом стоит необязательный знак, плюс или минус; при отсутствии знака число считается положительным.

Если символьная строка не соответствует этим критериям или значения должны округляться до заданного количества десятичных знаков, вызывайте функцию TO_NUMBER с маской форматирования.

Использование TO_NUMBER с маской форматирования

Применение функции TO_NUMBER с маской форматирования позволяет получить более разнообразные представления чисел. В табл. Б.1 (приложение Б) приведен полный список

всех поддерживаемых элементов маски форматирования. Например, местоположение разделителей групп и символ денежной единицы могут задаваться следующим образом:

```
a := TO_NUMBER('$123,456.78', 'L999G999D99');
```

Указывать в форматной строке точное количество цифр не обязательно. Функция `TO_NUMBER` позволяет задать в строке форматирования больше цифр, чем содержится в преобразуемом значении. Следующая строка кода также выполняется без ошибок:

```
a := TO_NUMBER('$123,456.78', 'L999G999G999D99');
```

Но если справа или слева от десятичной запятой значение содержит больше цифр, чем допускает маска форматирования, произойдет ошибка. Первое из следующих двух преобразований завершится ошибкой, поскольку строка содержит десять цифр слева от десятичной запятой, тогда как маска разрешает только девять. Второе преобразование завершается ошибкой из-за того, что строка содержит слишком много цифр справа от десятичной запятой:

```
a := TO_NUMBER('$1234,567,890.78', 'L999G999G999D99');
```

```
a := TO_NUMBER('$234,567,890.789', 'L999G999G999D99');
```

Элемент форматирования `0` обеспечивает вывод начальных нулей:

```
a := TO_NUMBER('001,234', '000G000');
```

Элемент `PR` распознает угловые скобки как обозначение отрицательного числа:

```
a := TO_NUMBER('<123.45>', '999D99PR');
```

Однако не все элементы форматирования предназначены для преобразования строк в числа. Например, элемент `RN`, предназначенный для вывода числа римскими цифрами, предназначен только для форматирования выводимой информации. Следующая попытка преобразования вызовет ошибку:

```
a := TO_NUMBER('cxxiii', 'rn');
```

Элемент `EEEE` тоже используется только для форматирования вывода. Для обратного преобразования он не нужен, поскольку функция `TO_NUMBER` правильно распознает числа в научной записи без дополнительных указаний:

```
a := TO_NUMBER('1.23456E-24');
```

Передача функции `TO_NUMBER` параметров NLS

Действие многих элементов форматирования определяется текущими установками параметров NLS. Например, элемент `G` представляет разделитель групп разрядов, но какой именно символ используется в качестве разделителя, зависит от текущего значения параметра `NLS_NUMERIC_CHARACTERS` в момент выполнения преобразования. Для просмотра текущих значений параметров можно запросить представление `NLS_SESSION_PARAMETERS`:

```
SQL> SELECT * FROM nls_session_parameters;
```

| PARAMETER | VALUE |
|------------------------|-----------|
| ----- | ----- |
| NLS_LANGUAGE | AMERICAN |
| NLS_TERRITORY | AMERICA |
| NLS_CURRENCY | \$ |
| NLS_ISO_CURRENCY | AMERICA |
| NLS_NUMERIC_CHARACTERS | ., |
| NLS_CALENDAR | GREGORIAN |
| NLS_DATE_FORMAT | DD-MON-RR |

Некоторые установки NLS по умолчанию зависят от других. Если присвоить параметру `NLS_TERRITORY` значение `AMERICA`, Oracle по умолчанию установит параметр `NLS_NUMERIC_CHARACTERS` равным `.,`. Это не мешает явно присвоить параметру

NLS_NUMERIC_CHARACTERS другое значение (например, с использованием команды ALTER SESSION).

Иногда отдельные параметры NLS требуется переопределить только на время вызова TO_NUMBER. В этом случае нужные установки задаются в этом вызове и действуют они исключительно для него. Например, в следующем примере при вызове TO_NUMBER задаются установки NLS, соответствующие NLS_TERRITORY=FRANCE:

```
a := TO_NUMBER('F123.456,78', 'L999G999D99',
               'NLS_NUMERIC_CHARACTERS=',',. '
               || ' NLS_CURRENCY='F' '
               || ' NLS_ISO_CURRENCY=FRANCE');
```

Строка параметров NLS получается слишком длинной, поэтому мы разбиваем ее на три строки, объединяемых оператором конкатенации, чтобы пример лучше смотрелся на странице. Обратите внимание на дублирование кавычек. Параметру NLS_NUMERIC_CHARACTERS требуется присвоить следующее значение:

```
NLS_NUMERIC_CHARACTERS=',, . '
```

Поскольку это значение вместе с кавычками включается в строку параметров NLS, каждую кавычку необходимо продублировать. В результате получается следующая строка:

```
'NLS_NUMERIC_CHARACTERS=',, . ' '
```

Функция TO_NUMBER позволяет задавать только три приведенных в данном примере параметра NLS. Было бы удобнее, если бы при вызове можно было использовать следующую запись:

```
a := TO_NUMBER('F123.456,78', 'L999G999D99', 'NLS_TERRITORY=FRANCE');
```

К сожалению, изменение параметра NLS_TERRITORY при вызове TO_NUMBER не поддерживается. Функция поддерживает только NLS_NUMERIC_CHARACTERS, NLS_CURRENCY и NLS_ISO_CURRENCY.



За подробной информацией о настройке параметров NLS обращайтесь к руководству «Oracle's Globalization Support Guide», которое является частью документации Oracle11g.

Передавать TO_NUMBER третий аргумент не рекомендуется — лучше полагаться на настройки сеанса, определяющие, как PL/SQL интерпретирует элементы маски форматирования (такие, как L, G и D). Вместо того чтобы жестко кодировать информацию в программах, нужно дать пользователю возможность задавать их на уровне сеанса.

Функция TO_CHAR

Функция TO_CHAR выполняет задачу, обратную функции TO_NUMBER: она преобразует число в его символьное представление. Используя необязательную маску форматирования, можно подробно указать, каким должно быть представление. Функция TO_CHAR вызывается следующим образом:

```
TO_CHAR(число [, формат [, параметры_nls]])
```

Здесь *число* — это число, которое требуется представить в символьной форме. Оно может относиться к любому из числовых типов PL/SQL: NUMBER, PLS_INTEGER, BINARY_INTEGER, BINARY_FLOAT, BINARY_DOUBLE, SIMPLE_INTEGER, SIMPLE_FLOAT или SIMPLE_DOUBLE. Параметр *формат* содержит необязательную маску форматирования, определяющую способ представления числа в символьной форме; необязательная строка *параметры_nls* содержит

значения параметров NLS. Ее можно применить для замещения текущих установок параметров NLS уровня сеанса.



Если вы хотите, чтобы результат был представлен в национальном наборе символов, используйте вместо `TO_CHAR` функцию `TO_NCHAR`. При этом помните, что строка форматирования числа должна быть представлена символами национального набора; в противном случае полученная строка будет состоять из символов «#».

Использование `TO_CHAR` без маски форматирования

Функция `TO_CHAR`, как и `TO_NUMBER`, может вызываться без маски форматирования:

```
DECLARE
  b VARCHAR2(30);
BEGIN
  b := TO_CHAR(123456789.01);
  DBMS_OUTPUT.PUT_LINE(b);
END;
```

Результат выглядит так:

123456789.01

В отличие от `TO_NUMBER` форма `TO_CHAR` особой пользы не приносит. Чтобы число лучше читалось, нужно задать как минимум разделитель групп разрядов.

Использование функции `TO_CHAR` с маской форматирования

При преобразовании числа в символьное представление функция `TO_CHAR` используется чаще всего с маской форматирования. Например, с ее помощью можно вывести денежную сумму:

```
DECLARE
  b VARCHAR2(30);
BEGIN
  b := TO_CHAR(123456789.01, 'L999G999G999D99');
  DBMS_OUTPUT.PUT_LINE(b);
END;
```

В локальном контексте США результат будет выглядеть так:

\$123,456,789.01

Элементы форматирования (см. табл. Б.1 в приложении Б) позволяют очень гибко определять формат символьного представления числа. Чтобы лучше понять, как они работают, стоит немного с ними поэкспериментировать. В следующем примере указано, что старшие разряды должны быть заполнены нулями, но при этом элемент форматирования `B` требует замены нулей пробелами. Данный элемент предшествует цифровым элементам (нулям), но следует за индикатором вывода знака денежной единицы `L`:

```
DECLARE
  b VARCHAR2(30);
  c VARCHAR2(30);
BEGIN
  b := TO_CHAR(123.01, 'LB000G000G009D99');
  DBMS_OUTPUT.PUT_LINE(b);
  c := TO_CHAR(0, 'LB000G000G009D99');
  DBMS_OUTPUT.PUT_LINE(c);
END;
```

Результат будет иметь следующий вид:

\$000,000,123.01

В примере выводится только одна строка, полученная после первого преобразования. В результате второго преобразования получается нуль, и элемент форматирования `V` заставляет `TO_CHAR` вернуть пустую строку, хотя в маске форматирования указано, что нулевые старшие разряды числа следует оставить. В качестве эксперимента попробуйте выполнить этот пример без элемента `V`.



Не все комбинации элементов форматирования являются допустимыми. Например, нельзя использовать сочетание `LRN`, которое выводит перед числом, записанным римскими цифрами, знак денежной единицы. Oracle не документирует такие нюансы, поэтому о некоторых из них можно узнать только на практике.

Элемент форматирования V

Элемент форматирования `V` достаточно необычен, чтобы его стоило упомянуть особо. Он позволяет масштабировать значение, а его действие лучше показать на примере (рис. 9.4).

123.45 ←999V9999

123.459999

1234599

1234500

1234500

- 1** Применение маски форматирования с элементом `V` к числу
- 2** Наложение числа на формат. Десятичная запятая выравнивается с позицией элемента `V`
- 3** Удаление элемента `V` и десятичной запятой
- 4** Дополнение числа нулями, чтобы итоговое количество значащих цифр получилось таким же, как в маске форматирования
- 5** Возвращение результата

Рис. 9.4. Элемент форматирования `V`

Зачем может понадобиться масштабирование? Рассмотрим простой пример. Стандартная единица сделки на бирже составляет 100 акций, и сообщая о реализованных на бирже акциях обычно говорят о количестве проданных пакетов по 100 акций. Поэтому 123 продажи означает 123 пакета по 100 акций, то есть 12 300 акций.

Следующий пример показывает, как использовать элемент `V` для масштабирования значения 123, с учетом того, что на самом деле оно представляет количество сотен:

```
DECLARE
  shares_sold NUMBER := 123;
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    TO_CHAR(shares_sold, '999G9V99')
  );
END;
```

Результат:

12,300

Заметьте, что в этом примере маска форматирования включает элемент `G`, определяющий местоположение разделителя групп (запятой), который может быть задан только слева от элемента `V`, что не всегда удобно. Следующая маска форматирования на первый взгляд выглядит вполне разумно:

```
TO_CHAR(123.45, '9G99V9G999');
```

Вы ожидаете, что результат будет отформатирован в виде 1,234,500, но элемент G не может располагаться справа от V. Можно использовать маску 9G99V9999 для получения результата 1,234500 или маску 999V9999 для получения 1234500, но оба эти результата выглядят не так, как нам хотелось бы.

Едва ли вы часто будете пользоваться элементом V, но знать о нем нужно.

Округление при преобразовании чисел в символьные строки

В том случае, когда при преобразовании символьной строки в число слева или справа от десятичной запятой получается больше цифр, чем допускает маска форматирования, происходит ошибка. Но при преобразовании числа в символьную строку ошибка возникает только при наличии лишних цифр слева от запятой. Если маска форматирования содержит меньше цифр после десятичной запятой, чем требуется для представления числа, число округляется до указанного количества цифр.

Если попытка преобразования завершается неудачей из-за того, что слева от запятой было слишком много цифр, функция `TO_CHAR` возвращает строку из символов «#». Например, следующее преобразование не будет выполнено, потому что число 123 не помещается в маску:

```
SQL> DECLARE
  2   b VARCHAR2(30); BEGIN
  3   b := TO_CHAR(123.4567, '99.99');
  4   DBMS_OUTPUT.PUT_LINE(b); END;
```

#####

Если же дробная часть числа не помещается в маску, происходит округление:

```
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE(TO_CHAR(123.4567, '999.99'));
  3   DBMS_OUTPUT.PUT_LINE(TO_CHAR(123.4567, '999')); END;
```

123.46
123

Цифры 5 и больше округляются в большую сторону, так что число 123,4567 округляется до 123,46, а цифры меньше 5 — в меньшую, поэтому 123,4xxx округляется до 123.

Пробелы при преобразовании чисел в символьные строки

В ходе преобразования числа в символьную строку функция `TO_CHAR` всегда оставляет место для знака «-», даже если число положительное.

```
DECLARE
  b VARCHAR2(30);
  c VARCHAR2(30);
BEGIN
  b := TO_CHAR(-123.4, '999.99');
  c := TO_CHAR(123.4, '999.99');
  DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || ' ' || TO_CHAR(LENGTH(b)));
  DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || ' ' || TO_CHAR(LENGTH(c)));
END;
```

Результат преобразования:

```
:-123.40 7
: 123.40 7
```

Обратите внимание: каждая строка имеет длину семь символов, хотя для положительного числа достаточно и шести. Начальный пробел применяется для выравнивания чисел

в столбцах. Но если по какой-то причине вам требуется вывести компактные числа без пробелов, это создаст проблемы.



Если для представления отрицательных чисел используются угловые скобки (в маске задан элемент PR), положительные числа дополняются одним начальным и одним завершающим пробелом.

Если числа, преобразованные в символьные данные, не должны содержать ни начальных, ни завершающих пробелов, существует несколько решений. Одно из них основано на использовании элемента форматирования TM, определяющего «минимальное» представление числа:

```
DECLARE
  b VARCHAR2(30);
  c VARCHAR2(30);
BEGIN
  b := TO_CHAR(-123.4, 'TM9');
  c := TO_CHAR(123.4, 'TM9');
  DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || TO_CHAR(LENGTH(b)));
  DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || TO_CHAR(LENGTH(c)));
END;
```

Результат:

```
:-123.4 6
:123.4 5
```

Этот метод удобен, но не позволяет задавать другие элементы форматирования. Так, нельзя задать формат TM999.99, чтобы число выводилось с двумя цифрами в дробной части. Если вам нужны другие элементы форматирования или если элемент TM не поддерживается в вашей версии PL/SQL, можно воспользоваться усечением результата:

```
DECLARE
  b VARCHAR2(30);
  c VARCHAR2(30);
BEGIN
  b := LTRIM(TO_CHAR(-123.4, '999.99'));
  c := LTRIM(TO_CHAR(123.4, '999.99'));
  DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || TO_CHAR(LENGTH(b)));
  DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || TO_CHAR(LENGTH(c)));
END;
```

Результат преобразования:

```
:-123.40 7
:123.40 6
```

Функция LTRIM была использована для удаления начальных пробелов и сохранения двух фиксированных цифр справа от десятичной запятой. Если же знак выводится справа от числа (например, с использованием элемента форматирования MI), можно воспользоваться функцией RTRIM. Если же при выводе используются элементы, влияющие на вывод с обеих сторон числа (например, PR), используется функция TRIM.

Передача параметров NLS функции TO_CHAR

По аналогии с функцией TO_NUMBER, функция TO_CHAR может получать в третьем параметре строку настроек NLS. Пример:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    TO_CHAR(123456.78, '999G999D99', 'NLS_NUMERIC_CHARACTERS='', ''')
  );
END;
```

Результат:

123.456,78

Таким способом можно задавать три параметра NLS: NLS_NUMERIC_CHARACTERS, NLS_CURRENCY и NLS_ISO_CURRENCY. Пример одновременного использования всех трех параметров приводился ранее, в разделе «Передача функции TO_NUMBER параметров NLS».

Функция CAST

Функция CAST применяется для преобразования чисел в строки, и наоборот. Синтаксис функции выглядит так:

CAST (*выражение AS тип_данных*)

В следующем примере функция CAST сначала используется для преобразования числа типа NUMBER в строку VARCHAR2, а затем символы этой строки преобразуются в соответствующее числовое значение:

```
DECLARE
  a NUMBER := -123.45;
  a1 VARCHAR2(30);
  b VARCHAR2(30) := '-123.45';
  b1 NUMBER;
  b2 BINARY_FLOAT;
  b3 BINARY_DOUBLE;
BEGIN
  a1 := CAST (a AS VARCHAR2);
  b1 := CAST (b AS NUMBER);
  b2 := CAST (b AS BINARY_FLOAT);
  b3 := CAST (b AS BINARY_DOUBLE);
  DBMS_OUTPUT.PUT_LINE(a1);
  DBMS_OUTPUT.PUT_LINE(b1);
  DBMS_OUTPUT.PUT_LINE(b2);
  DBMS_OUTPUT.PUT_LINE(b3);
END;
```

Результат выполнения:

```
-123.45
-123.45
-1.23449997E+002
-1.2345E+002
```

У функции CAST есть один недостаток: она не поддерживает маски форматирования чисел. С другой стороны, эта функция является частью стандарта ISO SQL — в отличие от функций TO_CHAR и TO_NUMBER. Если разработчику важно, чтобы программный код был полностью совместим со стандартом ANSI, используйте для преобразования чисел в строковые значения функцию CAST. В других случаях мы рекомендуем применять функции TO_CHAR и TO_NUMBER.



Поскольку PL/SQL не соответствует стандарту ISO, написать на этом языке полностью совместимый с указанным стандартом код невозможно. Таким образом, функция CAST в программах на PL/SQL становится лишней. Она востребована только в SQL-командах (SELECT, INSERT и т. д.), если они должны быть совместимы со стандартом ANSI.

Неявные преобразования

Преобразования между числами и строками можно выполнить еще одним способом: просто поручите эту работу PL/SQL. Такое преобразование называется *неявным*, так как оно не определяется явно в программном коде. Пример неявных преобразований:

```
DECLARE
  a NUMBER;
  b VARCHAR2(30);
BEGIN
  a := '-123.45';
  b := -123.45;
...
```

Как упоминалось в главе 7, неявные преобразования имеют ряд недостатков. Прежде всего, я считаю, что разработчик должен в полной мере контролировать свой код, а при использовании этого метода этот контроль отчасти утрачивается. Всегда лучше знать, когда и какое именно выполняется преобразование, поэтому желательно выполнять его явно. Если полагаться на неявное преобразование, то вы не сможете отследить, где и когда оно выполняется, и код станет менее эффективным. Кроме того, наличие явных преобразований делает код более понятным другим программистам.

Другой недостаток неявных преобразований заключается в том, что они могут нормально работать (по крайней мере на первый взгляд) в простых случаях, но их результат не всегда очевиден. Рассмотрим пример:

```
DECLARE
  a NUMBER;
BEGIN
  a := '123.400' || 999;
```

ОСТЕРЕГАЙТЕСЬ НЕЯВНЫХ ПРЕОБРАЗОВАНИЙ!

В разделе «Типы BINARY_FLOAT и BINARY_DOUBLE» этой главы я привел код (binary_performance.sql), использовавшийся для сравнения производительности BINARY_DOUBLE и NUMBER. В первой версии этого теста циклы для вычисления площади были запрограммированы следующим образом:

```
DECLARE
  bd BINARY_DOUBLE;
...
BEGIN
...
  FOR bd IN 1..1000000 LOOP
    bd_area := bd**2 * pi_bd;
  END LOOP;
...
```

Я был потрясен, когда результаты вдруг показали, что вычисления с NUMBER выполняются намного быстрее вычислений с BINARY_DOUBLE. Это было совершенно непостижимо, потому что я «знал», что операции с BINARY_DOUBLE выполняются на аппаратном уровне, а следовательно, просто обязаны работать быстрее операций с NUMBER.

Потом кто-то из сотрудников Oracle Corporation указал мне на мою ошибку: цикл FOR (в приведенном виде) неявно объявляет переменную цикла PLS_INTEGER с именем bd. Область действия нового объявления bd перекрывает блок цикла и замещает мое объявление bd в формате BINARY_DOUBLE. Кроме того, я записал константу в формате 2 (вместо 2d), из-за чего она интерпретировалась как NUMBER. Таким образом, значение bd сначала неявно преобразовывалось в NUMBER, затем возводилось в квадрат, и полученное значение NUMBER неявно снова преобразовывалось в BINARY_DOUBLE для умножения на pi_bd. Неудивительно, что результаты были настолько плохи! Подобные ловушки характерны для неявных преобразований.

Какое значение будет содержать переменная `a` после выполнения этого кода? Это зависит от того, как PL/SQL вычисляет выражение в правой части оператора присваивания. Если он сначала преобразует строку в число, мы получим следующий результат:

```
a := '123.400' || 999;
a := 123.4 || 999;
a := '123.4' || '999';
a := '123.4999';
a := 123.4999;
```

С другой стороны, если PL/SQL сначала преобразует число в строку, то результат будет таким:

```
a := '123.400' || 999;
a := '123.400' || '999';
a := '123.400999';
a := 123.400999;
```

И какой из двух результатов будет получен? Вы знаете? Даже если *знаете*, вряд ли другие программисты сразу догадаются об этом, читая ваш код. В данном случае лучше записать преобразование в явном виде:

```
a := TO_NUMBER('123.400' || TO_CHAR(999));
```

Кстати говоря, это выражение соответствует порядку обработки исходного выражения базой данных. Согласитесь, с явно выраженными преобразованиями его намного проще понять с первого взгляда.

Числовые операторы

PL/SQL реализует несколько операторов, используемых при работе с числами. Эти операторы перечислены в табл. 9.6 в порядке возрастания приоритетов. Операторы с более низким приоритетом выполняются до операторов с более высоким приоритетом.

За подробными описаниями операторов обращайтесь к справочнику Oracle *SQL Reference*.

Таблица 9.6. Числовые операторы и их приоритеты

| Оператор | Операция | Приоритет |
|----------------|---------------------------|-----------|
| ** | Возведение в степень | 1 |
| + | Тождество | 2 |
| - | Отрицание | 2 |
| * | Умножение | 3 |
| / | Деление | 3 |
| + | Сложение | 4 |
| - | Вычитание | 4 |
| = | Равно | 5 |
| < | Меньше чем | 5 |
| > | Больше чем | 5 |
| <= | Меньше либо равно | 5 |
| >= | Больше либо равно | 5 |
| <>, !=, ~=, ^= | Не равно | 5 |
| IS NULL | Проверка неопределенности | 5 |
| BETWEEN | Принадлежность диапазону | 5 |
| NOT | Логическое отрицание | 6 |
| AND | Конъюнкция | 7 |
| OR | Дизъюнкция | 8 |

Числовые функции

В PL/SQL реализовано множество функций для работы с числами. Мы уже рассматривали функции преобразования `TO_CHAR`, `TO_NUMBER`, `TO_BINARY_FLOAT` и `TO_BINARY_DOUBLE`. В нескольких ближайших разделах приведены краткие описания важнейших функций. За подробными описаниями конкретных функций обращайтесь к справочнику Oracle *SQL Reference*.

Функции округления и усечения

Существуют четыре числовые функции, выполняющие округление и усечение числовых значений: `CEIL`, `FLOOR`, `ROUND` и `TRUNC`. Выбор нужной функции для конкретной ситуации может вызвать затруднения, поэтому в табл. 9.7 приводятся их сравнительные описания, а на рис. 9.5 представлены результаты вызова всех четырех функций с разными значениями.

Таблица 9.7. Функции округления и усечения чисел

| Функция | Описание |
|--------------------|--|
| <code>CEIL</code> | Возвращает наименьшее целое число, большее либо равное заданному значению |
| <code>FLOOR</code> | Возвращает наибольшее целое число, меньшее либо равное заданному значению |
| <code>ROUND</code> | Выполняет округление числа. Положительное значение параметра определяет способ округления цифр, находящихся справа от запятой, а отрицательное — находящихся слева |
| <code>TRUNC</code> | Усекает число до заданного количества десятичных знаков, отбрасывая все цифры, находящиеся справа |

| | 1.75 | 1.3 | 55.56 | 55.56 | 10 | Входное значение |
|--------------|------|-----|-------|-------|----|---------------------|
| Функция | 0 | 0 | 1 | -1 | 2 | Количество разрядов |
| ROUND | 2 | 1 | 55.6 | 60 | 10 | |
| TRUNC | 1 | 1 | 55.5 | 50 | 10 | |
| FLOOR | 1 | 1 | 55 | 55 | 10 | |
| CEIL | 2 | 2 | 56 | 56 | 10 | |

Рис. 9.5. Функции округления и усечения

Тригонометрические функции

В PL/SQL поддерживаются все основные тригонометрические функции. При их использовании следует помнить, что углы задаются в радианах, а не в градусах. Преобразование радианов в градусы и наоборот выполняется по следующим формулам:

```
radians = pi * degrees / 180 -- Градусы в радианы
degrees = radians * 180 / pi -- Радианы в градусы
```

В PL/SQL нет отдельной функции для получения числа π , однако его можно получить косвенным методом:

```
ACOS (-1)
```

Арккосинус числа -1 равен значению π . Конечно, поскольку это число представляет собой бесконечную десятичную дробь, вы всегда будете работать с его приближенным значением. Для получения нужной точности можно округлить результат вызова `ACOS(-1)` до нужного количества позиций функцией `ROUND`.

Сводка числовых функций

В этом разделе представлены краткие описания всех встроенных функций PL/SQL. Там, где это возможно, функции перегружаются для разных числовых типов. Например:

○ ABS

Функция перегружена для типов `BINARY_DOUBLE`, `BINARY_FLOAT`, `NUMBER`, `SIMPLE_INTEGER`, `SIMPLE_FLOAT`, `SIMPLE_DOUBLE` и `PLS_INTEGER`, так как операция определения абсолютного значения применима как к вещественным, так и к целочисленным значениям.

○ BITAND

Функция перегружена для типов `PLS_INTEGER` и `INTEGER` (подтип `NUMBER`), так как операция `AND` может применяться только к целочисленным значениям.

○ CEIL

Функция перегружена для типов `BINARY_DOUBLE`, `BINARY_FLOAT` и `NUMBER`, поскольку функция `CEIL` не актуальна для целых чисел.

Чтобы узнать, для каких типов перегружена та или иная функция, запросите описание встроенного пакета `SYS.STANDARD`:

```
SQL> DESCRIBE SYS.STANDARD
```

```
FUNCTION CEIL RETURNS NUMBER
Argument Name          Type                      In/Out Default?
-----
N                      NUMBER                   IN
FUNCTION CEIL RETURNS BINARY_FLOAT
Argument Name          Type                      In/Out Default?
-----
F                      BINARY_FLOAT             IN
FUNCTION CEIL RETURNS BINARY_DOUBLE
Argument Name          Type                      In/Out Default?
-----
D                      BINARY_DOUBLE            IN
...
```

Почти все функции в следующем списке определяются во встроенном пакете `SYS.STANDARD`. Единственным исключением является функция `BIN_TO_NUM`. За полной документацией по отдельным функциям обращайтесь к справочнику Oracle *SQL Reference*.

ABS(n)

Возвращает абсолютное значение числа.

ACOS(n)

Возвращает арккосинус угла `n` из диапазона $[-1; 1]$. Возвращаемое функцией значение находится в пределах от 0 до π .

ASIN(n)

Возвращает арксинус угла `n` из диапазона $[-1; 1]$. Возвращаемое функцией значение находится в пределах от $-\pi/2$ до $\pi/2$.

ATAN(n)

Возвращает арктангенс угла `n` из диапазона $(-\infty; +\infty)$. Возвращаемое функцией значение находится в пределах от $-\pi/2$ до $\pi/2$.

ATAN2(n, m)

Возвращает арктангенс `n/m` для чисел `n` и `m` из диапазона $(-\infty; +\infty)$. Возвращаемое функцией значение находится в пределах от $-\pi/2$ до $\pi/2$.

BIN_TO_NUM(b1, b2,...bn)

Преобразует битовый вектор, представленный значениями от **b1** до **bn**, в число. Каждое из значений вектора должно быть равно либо 0, либо 1. Например, результат вызова **BIN_TO_NUM(1, 1, 0, 0)** равен 12.

BITAND(n, m)

Выполняет поразрядную операцию **AND** над битами двух положительных целых чисел. Например, вызов **BITAND(12, 4)** дает результат 4, то есть в значении 12 (двоичное 1100) установлен 4-й бит.

Вам будет проще работать с **BITAND**, если вы ограничитесь положительными целыми числами. Значения типа **PLS_INTEGER**, особенно удобного в сочетании с **BITAND**, позволяют хранить значения до 230; таким образом, в вашем распоряжении 30 битов для выполнения поразрядных операций.

CEIL(n)

Возвращает наименьшее целое число, которое больше либо равно заданному значению. В табл. 9.7 и рис. 9.5 функция **CEIL** сравнивается с другими числовыми функциями округления и усечения.

COS(n)

Возвращает косинус угла **n**, заданного в радианах. Если угол задается в градусах, то значение следует преобразовать в радианы (см. раздел «Тригонометрические функции»).

COSH(n)

Возвращает гиперболический косинус **n**. Если **n** — вещественное число, а **i** — мнимая единица, тогда связь между функциями **COS** и **COSH** выражается следующей формулой:

$$\cos(i * n) = \cosh(n)$$

EXP(n)

Возвращает число **e** в степени **n**, где **n** — аргумент функции. Число **e** (приблизительно равное 2,71828) является основанием натурального логарифма.

FLOOR(n)

Возвращает наибольшее целое число, которое меньше или равно заданному значению. В табл. 9.7 и рис. 9.5 функция **FLOOR** сравнивается с другими числовыми функциями округления и усечения.

GREATEST(n1, n2,...n3)

Возвращает наибольшее число во входном списке; например, результат вызова **GREATEST(1, 0, -1, 20)** равен 20.

LEAST(n1, n2,...n3)

Возвращает наименьшее число во входном списке; например, результат вызова **LEAST(1, 0, -1, 20)** равен -1.

LN(n)

Возвращает натуральный логарифм числа. Значение аргумента **n** должно быть больше 0 или равно ему. Если вызвать **LN** с отрицательным аргументом, выводится сообщение об ошибке.

LOG(b, n)

Возвращает логарифм заданного числа по указанному основанию. Значение аргумента n должно быть больше 0 или равно ему, а основание b должно быть больше 1. Если какой-либо из аргументов LOG не отвечает этим требованиям, выводится сообщение об ошибке.

MOD(n, m)

Возвращает остаток от деления n на m . Остаток вычисляется по формуле, эквивалентной $n - (m * \text{FLOOR}(n/m))$ при совпадении знаков n и m или $n - (m * \text{CEIL}(n/m))$ при различающихся знаках. Например, результат вызова MOD(10, 2.8) равен 1.6. Если аргумент m равен 0, возвращается значение n .

С помощью функции MOD можно быстро проверить число на четность или нечетность:

```
FUNCTION is_odd (num_in IN NUMBER) RETURN BOOLEAN
IS
BEGIN
    RETURN MOD (num_in, 2) = 1;
END;

FUNCTION is_even (num_in IN NUMBER) RETURN BOOLEAN
IS
BEGIN
    RETURN MOD (num_in, 2) = 0;
END;
```

NANVL(n, m)

Возвращает m , если n является NaN («не числом»); в противном случае возвращается n . Возвращаемое значение относится к числовому типу аргумента, обладающему наибольшим приоритетом в следующем порядке: BINARY_DOUBLE, BINARY_FLOAT или NUMBER.

POWER(n, m)

Возводит n в степень m . Если значение n отрицательно, то аргумент m должен быть целым числом. В следующем примере функция POWER используется для вычисления диапазона допустимых значений переменной типа PLS_INTEGER (от $-2^{31} - 1$ до $2^{31} - 1$):

```
POWER (-2, 31) - 1 .. POWER (2, 31) - 1
```

Результат:

```
-2147483648 .. 2147483647
```

REMAINDER(n, m)

Возвращает «псевдоостаток» от деления n на m . Значение вычисляется по следующей формуле:

$$n - (m * \text{ROUND}(n/m))$$

Например, результат вызова REMAINDER(10, 2.8) равен -1.2. Сравните с функцией MOD.

ROUND(n)

Возвращает значение n , округленное до ближайшего целого. Пример:

```
ROUND (153.46) --> 153
```


ROUND(n, m)

Возвращает значение n , округленное до m разрядов. Значение m может быть отрицательным: в этом случае функция **ROUND** отсчитывает позиции округления влево, а не вправо от десятичной запятой. Примеры:

```
ROUND (153.46, 1) --> 153.5  
ROUND (153, -1) --> 150
```

В табл. 9.7 и рис. 9.5 функция **ROUND** сравнивается с другими числовыми функциями округления и усечения.

SIGN(n)

Возвращает -1 , 0 или $+1$, если значение n меньше нуля, равно нулю или больше нуля соответственно.

SIN(n)

Возвращает синус угла n , заданного в радианах. Если угол задается в градусах, значение следует преобразовать в радианы (см. раздел «Тригонометрические функции»).

SINH(n)

Возвращает гиперболический синус n . Если n — вещественное число, а i — мнимая единица, тогда связь между функциями **SIN** и **SINH** выражается следующей формулой:

```
SIN (i * n) = i * SINH (n)  
SQRT(n)
```

Возвращает квадратный корень числа n , которое должно быть больше либо равно 0 . При отрицательном значении n выводится сообщение об ошибке.

TAN(n)

Возвращает тангенс угла n , заданного в радианах. Если угол задается в градусах, значение следует преобразовать в радианы (см. раздел «Тригонометрические функции»).

TANH(n)

Возвращает гиперболический тангенс n . Если n — вещественное число, а i — мнимая единица, тогда связь между функциями **TAN** и **TANH** выражается следующей формулой:

```
TAN (i * n) = i * TANH (n)
```

TRUNC(n)

Усекает значение n до целого числа. Например, результат вызова **TRUNC**(10.51) равен 10.

TRUNC(n, m)

Усекает значение n до m разрядов. Например, результат вызова **TRUNC**(10.789, 2) равен 10.78. Значение m может быть отрицательным: в этом случае функция **TRUNC** отсчитывает позиции усечения влево, а не вправо от десятичной запятой. Так, вызов **TRUNC**(1264, -2) дает значение 1200.

В табл. 9.7 и рис. 9.5 функция **CEIL** сравнивается с другими числовыми функциями округления и усечения.

10 Дата и время

Большинство приложений выполняют те или иные операции со значениями даты и времени. Работать с датами довольно сложно; кроме того что приходится иметь дело с жестким форматированием данных, существует множество правил определения их допустимых значений и проведения корректных вычислений (приходится учитывать високосные годы, национальные праздники и выходные, диапазоны дат и т. д.). К счастью, СУБД Oracle и PL/SQL предоставляют набор типов данных для хранения даты и времени в стандартном внутреннем формате.

Для любого значения даты или времени Oracle сохраняет некоторые (или все) из перечисленных составляющих: год, месяц, день, час, минуты, секунды, часовой пояс, смещение часового пояса в часах, смещение часового пояса в минутах.

Впрочем, поддержка типов даты и времени — только часть дела. Еще необходим язык, средства которого позволяют удобно и естественно работать с этими значениями. Oracle обеспечивает разработчиков исчерпывающим набором функций для выполнения всевозможных операций с датами и временем.

Типы данных даты и времени

В течение долгого времени для работы с датой и временем в Oracle поддерживался только тип `DATE`. В Oracle9i ситуация немного изменилась: появились три новых типа `TIMESTAMP` и два новых типа `INTERVAL`. Они предоставляют много новых полезных возможностей, одновременно улучшая совместимость Oracle со стандартом ISO SQL. Типы данных `INTERVAL` подробно рассматриваются позднее в этой главе, а пока остановимся на четырех основных типах даты/времени.

○ `DATE`

Хранит значение даты и времени с точностью до секунд. Не содержит информации часового пояса.

○ `TIMESTAMP`

Хранит значение даты и времени без информации о часовом поясе. Эквивалентен типу данных `DATE`, отличаясь от него лишь тем, что время хранится с точностью до миллиардной доли секунды.

○ `TIMESTAMP WITH TIME ZONE`

Хранит вместе со значением даты и времени информацию о часовом поясе с точностью до девяти десятичных позиций.

○ **TIMESTAMP WITH LOCAL TIME ZONE**

Хранит значение даты и времени с точностью до девяти десятичных позиций. Значения этого типа автоматически преобразуются между часовым поясом базы данных и местным (сеансовым) часовым поясом. При хранении в базе данных значения преобразуются к часовому поясу базы данных, а при выборке они преобразуются к местному (сеансовому) часовому поясу.

Разобраться во всех особенностях этих типов, особенно **TIMESTAMP WITH LOCAL TIME ZONE**, бывает непросто. Для примера рассмотрим использование типа **TIMESTAMP WITH LOCAL TIME ZONE** в календарном приложении для пользователей, работающих в разных часовых поясах. В качестве времени базы данных используется всеобщее скоординированное время UTC (Universal Coordinated Time — см. далее врезку «Всеобщее скоординированное время»). Пользователь Джонатан, живущий в Мичигане (Восточный часовой пояс, смещение от UTC составляет $-4:00$), запланировал проведение видеоконференции с 16:00 до 17:00 в четверг по своему местному времени. У Донны из Денвера (Горный часовой пояс, смещение составляет $-6:00$) конференция приходится на промежуток времени с 14:00 до 15:00 в четверг. У Селвы из Индии (смещение $+5:30$) конференция пройдет с 01:30 до 02:30 в пятницу. На рис. 10.1 показано, как время начала конференции изменяется при выборке из базы данных пользователями из разных часовых поясов.

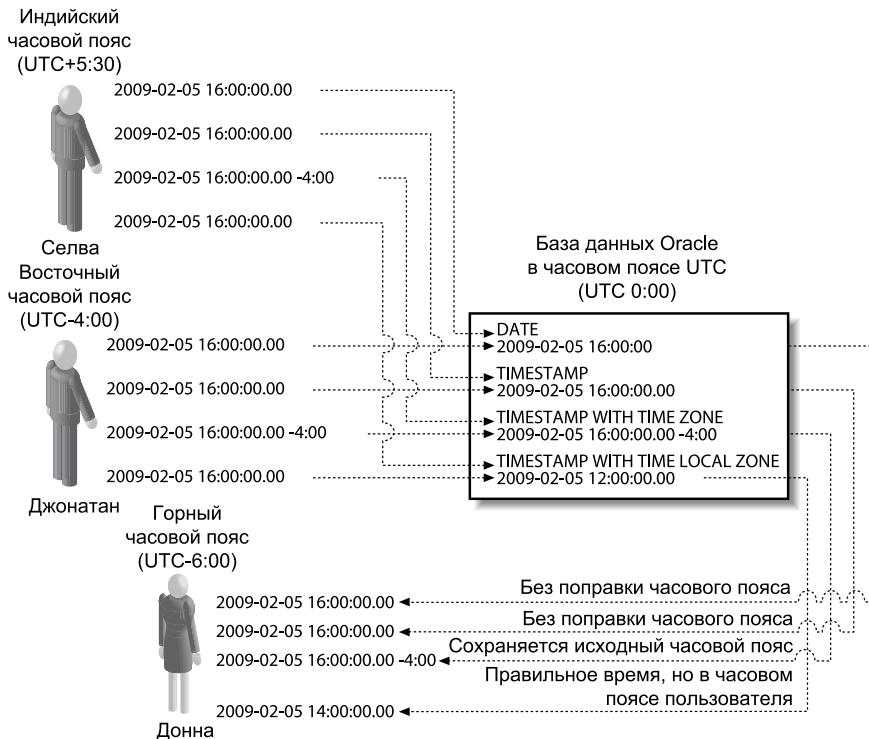


Рис. 10.1. Разные типы значений даты-времени

На рис. 10.1 пользователь Джонатан находится в Восточном часовом поясе с действием летнего времени, в котором время отстает от UTC на 4 часа (UTC-4:00). Джонатан вводит время начала собрания 16:00. Это значение преобразуется к часовому поясу базы данных (UTC) при вставке записи, и в базе данных сохраняется значение 20:00.

Донна находится в Денвере, где также действует летнее время; текущее время отстает от UTC (UTC-6:00). Когда Донна выбирает время начала встречи, значение преобразуется к сеансовому часовому поясу и отображается в формате 14:00. Селва находится в Индии, где летнее время не действует — индийское стандартное время смещено на 5,5 часа вперед от UTC (UTC+5:30). Когда Селва выбирает время начала встречи, значение преобразуется к сеансовому часовому поясу и выводится в формате 1:30.

Поручая преобразования часовых поясов типу данных `TIMESTAMP WITH LOCAL TIME ZONE`, вы избавляетесь от необходимости программирования сложных правил, связанных с часовыми поясами и летним временем (которое иногда изменяется, как это было в США в 2007 году), а заодно избавляете ваших пользователей от необходимости разбираться с преобразованиями. Правильное время будет предоставляться каждому пользователю просто и элегантно.

В одних случаях база данных должна автоматически изменять формат вывода времени, в других это не нужно. Если вы не хотите, чтобы формат значения времени изменялся в соответствии с сеансовыми настройками, используйте тип данных `TIMESTAMP` или `TIMESTAMP WITH TIME ZONE`.

ВСЕОБЩЕЕ СКООРДИНИРОВАННОЕ ВРЕМЯ

Всеобщее скоординированное время, обозначаемое сокращением UTC (Coordinated Universal Time), измеряется с применением высокоточных атомных часов и закладывает основу для мировой системы гражданского времени. Например, все часовые пояса определяются их смещением от UTC. Время UTC измеряется по атомному эталону и периодически регулируется через механизм корректировочных секунд для поддержания синхронизации с временем, определяемым по вращению Земли.

Возможно, вы также знакомы с временем по Гринвичскому меридиану, или GMT (Greenwich Mean Time). Как правило, в практическом контексте это обозначение эквивалентно UTC.

Почему выбрано сокращение UTC, а не CUT? Комитет по стандартизации не мог решить, стоит ли использовать английское сокращение CUT или французское TUC, поэтому сошлись на сокращении UTC, которое не соответствует ни одному языку.

За дополнительной информацией о UTC обращайтесь к документации Национального института стандартов и технологий (<http://www.nist.gov/pml/general/time/world.cfm>) и списку FAQ (<http://www.nist.gov/pml/div688/faq.cfm>).

Объявление переменных даты и времени

Синтаксис объявления переменной, представляющей дату и время, выглядит так:

```
имя_переменной [CONSTANT] тип [{:= | DEFAULT} исходное_значение]
```

Поле *тип* заменяется одним из следующих типов:

```
DATE  
TIMESTAMP [(точность)]  
TIMESTAMP [(точность)] WITH TIME ZONE  
TIMESTAMP [(точность)] WITH LOCAL TIME ZONE
```

Значение параметра *точность* определяет количество десятичных цифр, выделяемое для хранения долей секунды. По умолчанию оно равно 6, то есть время может отслеживаться

с точностью до 0,000001 секунды. Допускаются значения от 0 до 9, позволяющие сохранять время суток с высокой точностью.



Функции, возвращающие значения типа `TIMESTAMP` (например, `SYSTIME-
STAMP`), всегда возвращают данные с шестью цифрами точности.

Несколько примеров объявлений:

```
DECLARE
  hire_date TIMESTAMP (0) WITH TIME ZONE;
  todays_date CONSTANT DATE := SYSDATE;
  pay_date TIMESTAMP DEFAULT TO_TIMESTAMP('20050204','YYYYMMDD');
BEGIN
  NULL;
END;
/
```

Исходное значение задается либо при помощи функции преобразования (например, `TO_TIMESTAMP`), либо с использованием литерала даты/времени. Оба варианта описаны далее в разделе «Преобразования даты и времени».



Поведение переменной типа `TIMESTAMP(0)` идентично поведению переменной типа `DATE`.

Выбор типа данных

Естественно, при таком богатстве выбора хочется понять, из каких соображений следует выбирать тип данных для представления даты/времени в той или иной ситуации. В значительной степени выбор типа данных зависит от требуемой детализации:

- Чтобы хранить время с точностью до долей секунды, используйте один из типов `TIMESTAMP`.
- Чтобы время автоматически преобразовывалось между часовыми поясами базы данных и сеанса, используйте тип `TIMESTAMP WITH LOCAL TIME ZONE`.
- Чтобы отслеживать часовой пояс сеанса, в котором были введены данные, используйте тип `TIMESTAMP WITH TIME ZONE`.
- Вы можете использовать `TIMESTAMP` вместо `DATE`. `TIMESTAMP` без долей секунд занимает 7 байт, как и тип данных `DATE`. При хранении долей секунд он занимает 11 байт.

Также могут действовать и другие факторы:

- Используйте `DATE` в тех ситуациях, в которых необходимо сохранить совместимость с существующим приложением, написанным до появления типов данных `TIMESTAMP`.
- В общем случае рекомендуется использовать в коде PL/SQL типы данных, соответствующие типам используемых таблиц базы данных (или по крайней мере совместимые с ними). Например, дважды подумайте, прежде чем читать значение `TIMESTAMP` из таблицы в переменную `DATE`, потому что это может привести к потере информации (в данном случае — долей секунд, и возможно, часового пояса).
- Если вы работаете с версией старше Oracle9i Database, у вас нет выбора — придется использовать `DATE`.
- При сложении и вычитании годов и месяцев поведение функции `ADD_MONTHS`, работающей со значениями типа `DATE`, отличается от поведения интервальных арифме-

тических операций с типами **TIMESTAMP**. За дополнительной информацией по этому важному, но неочевидному вопросу обращайтесь к разделу «Когда используются типы **INTERVAL**».



Будьте осторожны при совместном использовании типов данных **DATE** и **TIMESTAMP**. Правила арифметических операций для этих типов сильно различаются. Будьте внимательны при использовании традиционных встроенных функций даты Oracle (таких, как **ADD_MONTHS** или **MONTHS_BETWEEN**) к значениям типов **TIMESTAMP**. См. далее раздел «Арифметические операции над значениями даты/времени».

Получение текущей даты и времени

В любом языке программирования важно знать, как получить текущую дату и время. Обычно это один из самых первых вопросов, возникающих при работе с датами в приложениях. До выхода Oracle8i в PL/SQL существовал только один способ получения даты и времени: функция **SYSDATE**. Начиная с Oracle9i, в вашем распоряжении появились все функции из табл. 10.1. От вас лишь требуется понять, как они работают и какие преимущества дает тот или иной вариант.

Таблица 10.1. Функции, возвращающие дату и время

| Функция | Часовой пояс | Тип возвращаемого значения |
|--------------------------|--------------------|---------------------------------|
| CURRENT_DATE | Сеанс | DATE |
| CURRENT_TIMESTAMP | Сеанс | TIMESTAMP WITH TIME ZONE |
| LOCALTIMESTAMP | Сеанс | TIMESTAMP |
| SYSDATE | Сервер базы данных | DATE |
| SYSTIMESTAMP | Сервер базы данных | TIMESTAMP WITH TIME ZONE |

Какую же функцию использовать в конкретной ситуации? Ответ зависит от нескольких факторов, которые, вероятно, стоит рассматривать в следующем порядке:

1. Если вы используете версию, предшествующую Oracle8i, или должны обеспечить совместимость с ней, выбор небогат: используйте **SYSDATE**.
2. Какое время вас интересует — вашего сеанса или сервера базы данных? В первом случае используйте функцию, возвращающую сеансовое время, а во втором — функцию, возвращающую часовой пояс базы данных.
3. Должен ли часовой пояс возвращаться в составе текущей даты и времени? Если должен, используйте функцию **SYSTIMESTAMP** или **CURRENT_TIMESTAMP**.

Если будет выбрана функция, возвращающая сеансовое время, проследите за тем, чтобы часовой пояс сеанса был задан правильно. Информацию о часовом поясе сеанса и базы данных можно получить при помощи функций **SESSIONTIMEZONE** и **DBTIMEZONE** соответственно. Чтобы получить время в часовом поясе базы данных, необходимо изменить часовой пояс сеанса на **DBTIMEZONE**, а затем использовать одну из сеансовых функций. Примеры использования этих функций:

BEGIN

```
DBMS_OUTPUT.PUT_LINE('Session Timezone='||SESSIONTIMEZONE);
DBMS_OUTPUT.PUT_LINE('Session Timestamp='||CURRENT_TIMESTAMP);
DBMS_OUTPUT.PUT_LINE('DB Server Timestamp='||SYSTIMESTAMP);
DBMS_OUTPUT.PUT_LINE('DB Timezone='||DBTIMEZONE);
EXECUTE IMMEDIATE 'ALTER SESSION SET TIME_ZONE=DBTIMEZONE';
DBMS_OUTPUT.PUT_LINE('DB Timestamp='||CURRENT_TIMESTAMP);
```

```
-- Возврат часового пояса сеанса к местному значению
EXECUTE IMMEDIATE 'ALTER SESSION SET TIME_ZONE=LOCAL';
END;
```

Результат:

```
Session Timezone=-04:00
Session Timestamp=23-JUN-08 12.48.44.656003000 PM -04:00
DB Server Timestamp=23-JUN-08 11.48.44.656106000 AM -05:00
DB Timezone=+00:00
DB Timestamp=23-JUN-08 04.48.44.656396000 PM +00:00
```

В этом примере сеанс начинается в Восточном часовом поясе (−4:00), тогда как сервер работает по времени Центрального часового пояса (−5:00), но при этом в самой базе данных используется время GMT (+00:00). Чтобы получить время в часовом поясе базы данных, мы сначала приводим часовой пояс сеанса в соответствие с часовым поясом базы данных, а затем вызываем функцию часового пояса сеанса `CURRENT_TIMESTAMP`. Наконец, часовой пояс сеанса снова возвращается к исходному местному значению.



Если вы используете эти функции для хронометража на уровне долей секунд, помните об ограничениях операционной системы и оборудования. Функции `CURRENT_TIMESTAMP`, `LOCALTIMESTAMP` и `SYSTIMESTAMP` возвращают значения в типах данных `TIMESTAMP WITH TIME ZONE` или `TIMESTAMP`, позволяющих определять время с разрешением до миллиардной доли секунды.

Все это, конечно, замечательно, но подумайте, откуда берется это время. База данных получает его от операционной системы в результате вызова `GetTimeOfDay` (Unix/Linux), `GetSystemTime` (Microsoft Windows) или другой аналогичной функции операционной системы. В свою очередь, операционная система зависит от оборудования. Если операционная система или используемое оборудование способны отслеживать время с точностью до сотых долей секунды, база данных не сможет возвращать результаты с большей точностью. Например, в системе Linux на процессоре Intel x86 вы сможете отслеживать время с точностью до миллионной доли секунды (шесть цифр), тогда как при работе базы данных в Microsoft Windows XP или Vista на том же оборудовании обеспечивается точность до тысячной доли секунды. Кроме того, хотя операционная система может возвращать временной штамп с шестью знаками, результат может не соответствовать реальной точности в одну микросекунду.

Что делать, если функции, возвращающей нужный тип данных, не существует — например, если время сервера нужно получить в переменной `TIMESTAMP`? Можно доверить неявное преобразование типов базе данных, но лучше воспользоваться явным преобразованием `CAST`. Пример:

```
DECLARE
  ts1 TIMESTAMP;
  ts2 TIMESTAMP;
BEGIN
  ts1 := CAST(SYSTIMESTAMP AS TIMESTAMP);
  ts2 := SYSDATE;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts1, 'DD-MON-YYYY HH:MI:SS AM'));
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts2, 'DD-MON-YYYY HH:MI:SS AM'));
END;
```

Результат:

```
24-FEB-2002 06:46:39 PM
24-FEB-2002 06:46:39 PM
```

Вызов `SYSTIMESTAMP` использует `CAST` для явного преобразования `TIMESTAMP WITH TIME ZONE` в `TIMESTAMP`. При вызове `SYSDATE` преобразование `DATE` в `TIMESTAMP` выполняется неявно.

ДЛЯ ЧЕГО НУЖНЫ ДВА ТИПА INTERVAL

Поначалу меня несколько удивило то, что Oracle ввела сразу два типа данных INTERVAL. Решение обрабатывать годы и месяцы отдельно от дней, часов минут и секунд выглядело довольно странно. Почему бы не создать единый тип данных INTERVAL, покрывающий все возможности? Однако оказалось, что за это нужно благодарить римского императора Юлия Цезаря (создателя юлианского календаря), определившего продолжительность большинства месяцев.

Два типа данных с разделительной линией на уровне месяцев пришлось определить потому, что месяц — это единственный компонент даты/времени с непостоянной продолжительностью. Возьмем интервал в 1 месяц и 30 дней. Какова его фактическая продолжительность? Меньше двух месяцев? Ровно два месяца? А может быть, больше? Если месяцем является январь, то через 30 дней уже наступит март, и получится интервал в 61 день, что уже несколько больше «двух месяцев». Если же этим месяцем является февраль, тогда интервал равен либо 59, либо 60 дням. Ну а если месяц — апрель, тогда интервал составит 60 дней.

Вместо того чтобы возиться со всеми этими сложностями, обусловленными разной продолжительностью месяцев и возникающими при сравнении интервалов, математических операциях с датами, а также при нормализации значений даты/времени, стандарт ISO SQL разбивает модель даты/времени на две части: в одной — год и месяц, во второй — все остальное. За дополнительной информацией по этому вопросу обращайтесь к книге A Guide to the SQL Standard (автор C.J. Date, издательство Addison-Wesley).

Типы данных INTERVAL

Типы, рассматривавшиеся до настоящего момента, представляют определенные моменты времени. Типы данных INTERVAL, появившиеся в Oracle9i, предназначены для сохранения и обработки временных *промежутков*. Чтобы получить более полное представление о них, вспомним, с какими значениями даты/времени мы сталкиваемся в повседневной жизни:

- **Момент** — временная точка, определенная с некоторой точностью. Например, когда мы собираемся встать утром в заданное время, это время представляет момент. Он может определяться с точностью до часа, до минуты и т. д. Все типы данных DATE и TIMESTAMP предназначены для определения моментов времени.
- **Интервал** — понятие, которое относится не к конкретной временной точке, а к определенному количеству времени. В повседневной жизни мы постоянно имеем дело с временными интервалами: работаем по восемь часов, обедаем в течение часа (если бы!) и т. д. Для представления интервалов используются два типа данных INTERVAL.
- **Период** — интервал, который начинается и заканчивается в заданный момент. Например, если вы встали в 8:00 и работали в течение восьми часов, то 8-часовой интервал времени, начинающийся в 8:00, можно считать периодом. В Oracle нет специализированного типа данных для непосредственной поддержки периодов, как нет его и в стандарте SQL.

Oracle поддерживает два типа данных INTERVAL. Оба типа были введены в Oracle9i, и оба соответствуют стандарту ISO SQL:

- INTERVAL YEAR TO MONTH — позволяет определить интервал времени в годах и месяцах;
- INTERVAL DAY TO SECOND — позволяет определить интервал времени в днях, часах, минутах и секундах (с долями секунд).

Объявление интервальных переменных

По сравнению с другими объявлениями переменных PL/SQL синтаксис объявлений переменных обоих типов INTERVAL несколько необычен. Помимо того, что имена этих типов состоят из нескольких слов, для них задается не одно, а два значения, определяющих точность:

имя_переменной INTERVAL YEAR [(точность_лет)] TO MONTH

или

имя_переменной INTERVAL DAY [(точность_дней)] TO SECOND [(точность_долей_секунды)]

Здесь *имя_переменной* — имя объявляемой переменной INTERVAL; *точность_лет* — количество цифр (от 0 до 4), выделенное для представления количества лет (по умолчанию 2); *точность_дней* — количество цифр (от 0 до 9), выделенное для представления количества дней (по умолчанию 2); *точность_долей_секунды* — количество цифр (от 0 до 9), выделенное для представления количества долей секунды (по умолчанию 6).

О точности значений интервалов, как правило, можно не беспокоиться. Значения типа INTERVAL YEAR TO MONTH всегда нормализуются таким образом, что количество месяцев лежит в диапазоне от 0 до 11. Фактически Oracle не позволяет задать месяц значением больше 11; интервал в 1 год и 13 месяцев должен быть выражен как 2 года и 1 месяц. Значение параметра *точность_лет* устанавливает максимальный размер интервала типа INTERVAL YEAR TO MONTH. Аналогичным образом значение параметра *точность_дней* устанавливает максимальный размер интервала типа INTERVAL DAY TO SECOND.

Точность для часов, минут и секунд для значения типа INTERVAL DAY TO SECOND не нужно задавать по той же причине, по которой не задается точность для месяцев значения типа INTERVAL YEAR TO MONTH. Интервалы INTERVAL DAY TO SECOND всегда нормализуются таким образом, что значения часов, минут и секунд находятся в естественных диапазонах: 0–23 часа, 0–59 минут и 0–59 секунд (за исключением долей секунды).

Доли секунды указываются потому, что значения типа INTERVAL DAY TO SECOND могут определять интервалы с указанной точностью до долей секунды. Значения типа INTERVAL YEAR TO MONTH не могут содержать долей месяца, и последние для них не задаются.

Когда используются типы INTERVAL

Используйте типы данных INTERVAL во всех случаях, когда вам потребуется обрабатывать промежутки времени. В этом разделе приведены два примера; хочется верить, что они вызовут у вас интерес и помогут представить, как эти типы данных могли бы использоваться в создаваемых вами системах.

Вычисление разности между двумя значениями даты/времени

Типы INTERVAL удобно использовать для вычисления разности между двумя значениями даты/времени. В следующем примере вычисляется срок работы сотрудника:

```
/* Файл в Сети: interval_between.sql */
DECLARE
    start_date TIMESTAMP;
    end_date TIMESTAMP;
    service_interval INTERVAL YEAR TO MONTH;
    years_of_service NUMBER;
    months_of_service NUMBER;
```

продолжение ➤

```

BEGIN
  -- Обычно начальная и конечная даты загружаются из базы данных.
  start_date := TO_TIMESTAMP('29-DEC-1988','dd-mon-yyyy');
  end_date := TO_TIMESTAMP('26-DEC-1995','dd-mon-yyyy');

  -- Определение и вывод количества отработанных лет и месяцев:
  service_interval := (end_date - start_date) YEAR TO MONTH;
  DBMS_OUTPUT.PUT_LINE(service_interval);

  -- Новая функция EXTRACT выделяет отдельные компоненты года и месяца.
  years_of_service := EXTRACT(YEAR FROM service_interval);
  months_of_service := EXTRACT(MONTH FROM service_interval);
  DBMS_OUTPUT.PUT_LINE(years_of_service || ' years and '
                        || months_of_service || ' months');
END;
```

Непосредственное вычисление количества лет и месяцев работы выполняется в следующей строке:

```
service_interval := (end_date - start_date) YEAR TO MONTH;
```

Здесь `YEAR TO MONTH` — часть синтаксиса выражения, возвращающего интервал. Подробнее о нем рассказывается далее в этой главе. Как видите, вычисление продолжительности интервала сводится к простому вычитанию одной даты из другой. Без типа данных `INTERVAL` нам пришлось бы программировать вычисления самостоятельно:

```

months_of_service := ROUND(months_between(end_date, start_date));
years_of_service := TRUNC(months_of_service/12);
months_of_service := MOD(months_of_service,12);
```

Решение, в котором не используется тип данных `INTERVAL`, оказывается более сложным как для программирования, так и для понимания кода.



Тип `INTERVAL YEAR TO MONTH` выполняет округление значений, и вы должны знать о возможных последствиях этой операции. За подробностями обращайтесь к разделу «Арифметические операции над значениями даты/времени».

Обозначение периода времени

В этом примере анализируется работа конвейерной сборки. Важной метрикой эффективности является время, необходимое для сборки каждого продукта. Сокращение этого интервала повышает эффективность работы конвейера, поэтому начальство желает постоянно контролировать его продолжительность. В нашем примере каждому продукту присваивается контрольный номер, используемый для его идентификации в процессе сборки. Информация хранится в следующей таблице:

```

TABLE assemblies (
  tracking_id NUMBER NOT NULL,
  start_time  TIMESTAMP NOT NULL,
  build_time  INTERVAL DAY TO SECOND
);
```

Также нам понадобится функция `PL/SQL`, возвращающая время сборки для заданного идентификатора `tracking_id`. Значение вычисляется вычитанием текущего времени из времени начала сборки. Арифметические операции с датами более подробно рассматриваются позднее в этой главе. Функция вычисления времени сборки:

```

FUNCTION calc_build_time (
  esn IN assemblies.tracking_id%TYPE
)
```

```
RETURN DSINTERVAL_UNCONSTRAINED
IS
  start_ts assemblies.start_time%TYPE;
BEGIN
  SELECT start_time INTO start_ts FROM assemblies
  WHERE tracking_id = esn;
  RETURN LOCALTIMESTAMP-start_ts;
END;
```

При передаче интервалов программам PL/SQL и из них необходимо использовать ключевое слово `UNCONSTRAINED` (см. далее раздел «Типы данных INTERVAL без ограничений»). Хранение времени сборки в таблице упрощает анализ данных. Мы можем легко определить минимальное, максимальное и среднее время сборки при помощи простых функций SQL, а также находить ответы на вопросы «Выполняется ли сборка по понедельникам быстрее, чем по вторникам?» или «Какая смена работает более производительнее, первая или вторая?» Впрочем, я забегаю вперед. Этот тривиальный пример просто демонстрирует основные концепции интервалов. Ваша задача как программиста — найти творческое применение этим концепциям.

Преобразование даты и времени

Теперь, когда вы имеете представление о типах данных Oracle, предназначенных для работы с датой и временем, давайте посмотрим, как присваивать даты значениям переменных этих типов и как использовать такие значения в дальнейшем. Дата и время в понятной для человека форме представляются в виде символьных строк наподобие «5 марта 2009 года» и «10:30», так что речь пойдет о преобразовании значений даты/времени — приведении символьных строк в соответствие с внутренним представлением Oracle, и наоборот.

PL/SQL проверяет и хранит даты от 1 января 4712 года до н. э. до 31 декабря 9999 года н. э. (В документации Oracle указана максимальная дата 31 декабря 4712 года; чтобы проверить диапазон дат в вашей версии, запустите сценарий `showdaterange.sql` с сайта издательства O'Reilly.) Если ввести дату без времени (во многих приложениях не требуется отслеживать время, поэтому PL/SQL позволяет его не задавать), то время в таком значении по умолчанию будет равно полуночи (00:00:00).

База данных Oracle способна интерпретировать практически все известные форматы даты и времени. Эта возможность основана на использовании *маски форматирования даты*, которая представляет собой строку специальных символов, определяющих формат даты для Oracle.

В следующем разделе мы рассмотрим несколько разных масок форматирования даты. Полный список всех элементов форматирования масок приводится в приложении В.

Преобразование строк в даты

Первой практической задачей, с которой вы столкнетесь при работе с датами, будет присваивание даты (или времени) переменной PL/SQL. Операция выполняется путем преобразования даты/времени во внутренний формат базы данных. Такие преобразования выполняются либо неявно, присваиванием символьной строки переменной типа даты/времени, либо явно, с помощью встроенных функций Oracle.

Неявное преобразование — вещь рискованная, и мы не рекомендуем на него полагаться. Пример неявного преобразования символьной строки при ее присваивании переменной типа `DATE`:

```
DECLARE
    birthdate DATE;
BEGIN
    birthdate := '15-Nov-1961';
END;
```

Такое преобразование зависит от установленного значения `NLS_DATE_FORMAT`. Оно будет успешно работать до тех пор, пока администратор базы данных не решит изменить это значение. Как только он это сделает, код перестанет функционировать. Нарушит работу кода и изменение значения параметра `NLS_DATE_FORMAT` на уровне сеанса.

Вместо того чтобы полагаться на неявные преобразования и установленное значение параметра `NLS_DATE_FORMAT`, лучше преобразовывать даты явно, с помощью встроенных функций Oracle — таких, как `TO_DATE`:

```
DECLARE
    birthdate DATE;
BEGIN
    birthdate := TO_DATE('15-Nov-1961', 'dd-mon-yyyy');
END;
```

Обратите внимание на передачу форматной строки `'dd-mon-yyyy'` во втором параметре вызова `TO_DATE`. Форматная строка управляет интерпретацией символов первого параметра функцией `TO_DATE`.

PL/SQL поддерживает следующие функции преобразования строк в дату и время:

- `TO_DATE(строка[, маска[, язык_nls]])` — преобразует символьную строку в значение типа `DATE`.
- `TO_DATE(число[, маска[, язык_nls]])` — преобразует строку, содержащую Юлианскую дату, в значение типа `DATE`.
- `TO_TIMESTAMP(строка[, маска[, язык_nls]])` — преобразует символьную строку в значение типа `TIMESTAMP`.
- `TO_TIMESTAMP_TZ(строка[, маска[, язык_nls]])` — преобразует символьную строку в значение типа `TIMESTAMP WITH TIME ZONE`. Функция также используется для преобразования к типу `TIMESTAMP WITH LOCAL TIME ZONE`.

Эти функции не только ясно показывают, что в коде выполняется преобразование, но и позволяют точно задать используемый формат даты/времени.



Вторая версия `TO_DATE` может использоваться только с маской форматирования J (Юлианская дата, то есть количество дней, прошедших с 1 января 4712 года до н. э.). Только в этом варианте использования `TO_DATE` в первом параметре может передаваться число.

В остальных случаях параметры имеют следующий смысл:

- *строка* — преобразуемая строковая переменная, литерал, именованная константа или выражение;
- *маска* — маска форматирования, используемая для преобразования строки. По умолчанию в качестве маски используется значение параметра `NLS_DATE_FORMAT`.
- *язык_nls* — необязательное обозначение языка, который должен использоваться для интерпретации имен и сокращений месяцев и дней в строке, в формате `'NLS_DATE_LANGUAGE=язык'`.

Здесь *язык* — один из языков, распознаваемых вашим экземпляром базы данных. За информацией о допустимых языках обращайтесь к руководству *Oracle Globalization Support Guide*.

Форматные элементы (см. приложение В) применяются при использовании функций семейства `TO_`. Например, следующие вызовы `TO_DATE` и `TO_TIMESTAMP` преобразуют символьные строки разных форматов в значения `DATE` и `TIMESTAMP`:

```
DECLARE
  dt DATE;
  ts TIMESTAMP;
  tstz TIMESTAMP WITH TIME ZONE;
  tsltz TIMESTAMP WITH LOCAL TIME ZONE;
BEGIN
  dt := TO_DATE('12/26/2005', 'mm/dd/yyyy');
  ts := TO_TIMESTAMP('24-Feb-2002 09.00.00.50 PM');
  tstz := TO_TIMESTAMP_TZ('06/2/2002 09:00:00.50 PM EST',
    'mm/dd/yyyy hh:mi:ssxff AM TZD');
  tsltz := TO_TIMESTAMP_TZ('06/2/2002 09:00:00.50 PM EST',
    'mm/dd/yyyy hh:mi:ssxff AM TZD');
  DBMS_OUTPUT.PUT_LINE(dt);
  DBMS_OUTPUT.PUT_LINE(ts);
  DBMS_OUTPUT.PUT_LINE(tstz);
  DBMS_OUTPUT.PUT_LINE(tsltz);
END;
```

Результат:

```
26-DEC-05
24-FEB-02 09.00.00.500000 PM
02-JUN-02 09.00.00.500000 PM -05:00
02-JUN-02 09.00.00.500000 PM
```

Обратите внимание на представление долей секунд (.50) и на использование маски `XFF`. Элемент форматирования `X` определяет местоположение десятичного разделителя (в данном случае точки), отделяющего количество целых секунд от дробной части. Можно было бы просто поставить в этом месте точку (`.FF`), но мы предпочли воспользоваться символом `X`, поскольку он указывает, что на этом месте должен находиться разделитель, соответствующей текущему значению параметра `NLS_TERRITORY`.

Ошибки Oracle из диапазона от `ORA-01800` до `ORA-01899` связаны с преобразованиями дат. Чтобы узнать о некоторых тонкостях обработки дат, просмотрите документацию по этим ошибкам и ознакомьтесь с описаниями их причин. Вот лишь некоторые из них:

- Литерал даты, переданный `TO_CHAR` для преобразования в дату, не может быть длиннее 220 символов.
- Одна маска форматирования не может одновременно содержать элемент даты юлианского календаря (`J`) и элемент дня года (`DDD`).
- Маска не может содержать несколько элементов для одного компонента даты/времени. Например, маска `YYYY-YY-YY-DD-MM` недопустима, потому что она включает два элемента года, `YYYY` и `YY`.
- 24-часовой формат времени (`HH24`) не может использоваться в маске одновременно с 12-часовым.

Как показывают предыдущие примеры, функция `TO_TIMESTAMP_TZ` может преобразовывать символьные строки с информацией часового пояса. И хотя часовые пояса на первый взгляд выглядят просто, в действительности все намного сложнее — вы убедитесь в этом в разделе «Часовые пояса».

Преобразование даты в строку

Присвоить переменной значение типа даты/времени — только полдела. Еще нужно прочитать его и представить в понятной для человека форме. Для этого в Oracle существует функция `TO_CHAR`.

Функция `TO_CHAR` предназначена для преобразования значения даты/времени в строку переменной длины. Она применяется и для типа `DATE`, и для всех типов семейства `TIMESTAMP`. Кроме того, она выполняет преобразование чисел в символьные строки (см. главу 9). Синтаксис функции `TO_CHAR`:

```
FUNCTION TO_CHAR
  (входная_дата IN DATE
    [, маска IN VARCHAR2
    [, язык_nls IN VARCHAR2]])
RETURN VARCHAR2
```

Здесь *входная_дата* — символьная строка, представляющая значение даты/времени; *маска* — строка из одного или нескольких элементов формата; *язык_nls* — строка, задающая язык отображения даты. Параметры *маска* и *язык_nls* не являются обязательными.



Чтобы результирующая строка была представлена символами национального набора, вместо функции `TO_CHAR` следует использовать функцию `TO_NCHAR`. Но имейте в виду, что в этом случае маска форматирования тоже должна быть задана в символах национального набора. В противном случае произойдет ошибка.

Если маска не задана, по умолчанию используется формат даты, установленный для базы данных. Это формат `'DD-MON-RR'`, если только администратор не задал другой формат, изменив значение параметра `NLS_DATE_FORMAT`. Как упоминалось ранее в этой главе, лучше не зависеть от неявных преобразований даты. Изменения серверных настроек `NLS` (а в клиентском коде — изменения клиентских настроек `NLS`) приводят к появлению логических ошибок, если ваш код зависит от неявных преобразований. Если приложение всегда выполняет преобразования явно, такие ошибки исключены.

Несколько примеров использования функции `TO_CHAR` для преобразования даты:

- Обратите внимание на два пробела между месяцем и днем, а также на начальный ноль перед номером месяца:

```
TO_CHAR (SYSDATE, 'Month DD, YYYY') --> 'February 05, 1994'
```

- Для подавления пробелов и нулей используется элемент `FM`:

```
TO_CHAR (SYSDATE, 'FMMonth DD, YYYY') --> 'February 5, 1994'
```

- Обратите внимание на различие регистра в обозначении месяца в следующих двух примерах (при работе с форматами данных Oracle вы получаете в точности то, что запрашиваете):

```
TO_CHAR (SYSDATE, 'MON DDth, YYYY') --> 'FEB 05TH, 1994'
```

```
TO_CHAR (SYSDATE, 'fmMon DDth, YYYY') --> 'Feb 5TH, 1994'
```

- На элемент форматирования `TH` не распространяются правила задания регистра. Даже если ввести его в нижнем регистре (`th`), то в выходной строке база данных будет использовать `TH`.

- Вывод дня года, дня месяца и дня недели для заданной даты:

```
TO_CHAR (SYSDATE, 'DDD DD D ') --> '036 05 7'
```

```
TO_CHAR (SYSDATE, 'fmDDD fmDD D ') --> '36 05 7'
```

- Нетривиальное форматирование для построения отчета:

```
TO_CHAR (SYSDATE, '"In month "RM" of year "YEAR"')
```

```
--> 'In month II of year NINETEEN NINETY FOUR'
```

- Для переменных типа `TIMESTAMP` время можно задавать с точностью до миллисекунд:

```
TO_CHAR (A_TIMESTAMP, 'YYYY-MM-DD HH:MI:SS.FF AM TZh:TzM')
```

```
--> значение вида 2002-02-19 01:52:00.123457000 PM -05:00
```

Будьте внимательны при работе со значениями, включающими доли секунд. В маске форматирования доли секунды представляются элементом FF, и кто-то решит, что количество букв F должно соответствовать количеству десятичных цифр в выходной строке. Но это не так! Для обозначения от 1 до 9 десятичных цифр следует использовать конструкции FF1–FF9. Например, в следующем блоке конструкция FF6 запрашивает вывод с точностью до шести десятичных цифр:

```
DECLARE
    ts TIMESTAMP WITH TIME ZONE;
BEGIN
    ts := TIMESTAMP '2002-02-19 13:52:00.123456789 -5:00';
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts, 'YYYY-MM-DD HH:MI:SS.FF6 AM TZN:TZM'));
END;
```

Результат:

```
2002-02-19 01:52:00.123457 PM -05:00
```

Обратите внимание на выполненное округление. Во входных данных было указано количество секунд 00.123456789. Значение было округлено (не усечено, а именно округлено) до шести цифр: 00,123457.

Ничего не стоит по ошибке задать неверный формат даты, и с появлением типа данных **TIMESTAMP** вероятность этого даже увеличилась. Некоторые элементы форматирования, использующиеся со значениями типа **TIMESTAMP**, не могут применяться со значениями типа **DATE**. Например, если попытаться преобразовать значения типа **DATE** в символьную строку с помощью элементов FF, TZN и TZM, получится следующее:

```
DECLARE
    dt DATE;
BEGIN
    dt := SYSDATE;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(dt, 'YYYY-MM-DD HH:MI:SS.FF AM TZN:TZM'));
END;
```

Результат будет таким:

```
dt := SYSDATE;
*
ORA-01821: date format not recognized
ORA-06512: at line 5
```

Сообщение об ошибке (формат даты не распознается) только сбивает с толку. Формат даты абсолютно нормален — проблема в том, что он применяется не к тому типу данных. Поэтому при получении такого сообщения следует проверить не только формат даты, но и тип данных, который вы собираетесь преобразовывать.

Часовые пояса

Возможное присутствие информации о часовом поясе несколько усложняет использование функции **TO_TIMESTAMP_TZ** по сравнению с функциями **TO_DATE** и **TO_TIMESTAMP**. Информация о часовом поясе задается одним из следующих способов:

- Как положительное или отрицательное смещение в часах и минутах относительно времени UTC; например, значение –5:00 эквивалентно Восточному стандартному времени США. Значения смещения должны находиться в диапазоне от –12:59 до +13:59. (Примеры записи смещения уже приводились ранее в этой главе.)
 - В виде названия часового пояса, например: US/Eastern, US/Pacific и т. д.
 - В виде сочетания названия и сокращения часового пояса, например: US/Eastern EDT.
- Рассмотрим несколько примеров. Начнем с простейшего случая, в котором часовой пояс вообще не указан:

```
TO_TIMESTAMP_TZ ('12312005 083015.50', 'MMDYYYYY HHMISS.FF')
```

Дата и время здесь определены как 31 декабря 1998 года, 8 часов 30 минут 15 с половиной секунд до полудня. Поскольку часовой пояс не указан, Oracle считает, что время относится к текущему часовому поясу. Если часовой пояс намеренно не указан, то программа получается менее выразительной, чем следует. Если приложение должно использовать часовой пояс сеанса (вместо явно заданного часового пояса), лучше сначала определить часовой пояс сеанса при помощи функции `SESSIONTIMEZONE`, а затем явно использовать его при вызове функции `TO_TIMESTAMP_TZ`. Явное выражение ваших намерений поможет разработчику (которым можете быть вы сами) разобраться в вашем коде два года спустя, при добавлении новых функций или исправлении ошибок.

ДАТА ИЛИ ВРЕМЯ?

Учтите, что каждое значение даты/времени содержит как дату, так и время. Стоит вам забыть об этом, и в вашем коде появятся ошибки. Рассмотрим пример: допустим, я написал код PL/SQL, который должен быть выполнен 1 января 2015 года:

```
IF SYSDATE = TO_DATE('1-Jan-2015', 'dd-Mon-yyyy')
THEN
    Apply2015PriceChange;
END IF;
```

Запускаемая процедура должна изменить цены на предстоящий год, но вероятность добиться желаемого результата минимальна. Блок кода должен быть выполнен ровно в полночь, с точностью до секунды; дело в том, что `SYSDATE` вместе с датой возвращает время. Чтобы блок работал так, как задумано, следует усечь значение, возвращаемое функцией `SYSDATE`, до полуночи соответствующего дня:

```
IF TRUNC(SYSDATE) = TO_DATE('1-Jan-2015', 'dd-Mon-yyyy');
```

Теперь в обеих сторонах сравнения указывается время суток, но этим временем является полночь. Функция `TO_DATE` также возвращает время суток, которое по умолчанию соответствует полуночи (то есть 00:00:00). Итак, в какое бы время 1 января 2015 года ни был выполнен этот блок, сравнение будет выполнено успешно, и процедура `Apply2009PriceChange` будет выполнена.

Функцию `TRUNCATE` также удобно использовать для удаления времени из временных штампов.

В следующем примере часовой пояс задается смещением в часах и минутах относительно UTC. Обратите внимание на элементы `TZH` и `TZM`, отмечающие, где во входной строке указывается смещение времени в часах и минутах:

```
TO_TIMESTAMP_TZ ('1231200 083015.50 -5:00', 'MMDDYY HHMISS.FF TZH:TZM')
```

В этом примере значение даты/времени интерпретируется как Восточное стандартное время США (независимо от часового пояса сеанса).

Следующий пример показывает, как задать часовой пояс по имени региона. В нем задается регион `EST`, что соответствует Восточному времени в Соединенных Штатах. Обозначение `TZR` в маске форматирования указывает местоположение имени региона во входной строке:

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00 EST',
                  'dd-Mon-yyyy hh:mi:ss TZR')
```

Этот пример интересен тем, что он представляет не Восточное стандартное время, а просто Восточное время. Разница между ними заключается в том, что термин Восточное время может обозначать как Восточное стандартное время, так и Восточное летнее

время — в зависимости от того, действует ли переход на летнее время. Я специально сформулировал этот пример так, чтобы он был неоднозначным. 2 ноября 2014 года — это дата завершения действия Восточного летнего времени, когда время 2:00 превращается в 1:00. Поэтому в указанный день 1:30 наступает дважды: по Восточному летнему и по Восточному стандартному времени. Так какое же время имеется в виду при определении 1:30 2 ноября 2014 года?

Названия региона недостаточно для различения стандартного и летнего времени. Для устранения этой неоднозначности необходимо задать сокращение часового пояса, как это сделано в двух следующих примерах. Восточное летнее время обозначается сокращением EDT (Eastern Daylight Time):

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00.00 US/Eastern EDT',  
                 'dd-Mon-yyyy hh:mi:ssx' || 'TZR TZD')
```



Если параметр сеанса `ERROR_ON_OVERLAP_TIME` равен `TRUE` (по умолчанию используется значение `FALSE`), база данных будет выдавать ошибку при задании неоднозначного времени.

Восточное стандартное время обозначается сокращением EST (Eastern Standard Time):

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00.00 US/Eastern EST',  
                 'dd-Mon-yyyy hh:mi:ssx' || 'TZR TZD')
```

Для предотвращения неоднозначности рекомендуется либо задавать смещение часового пояса в часах и минутах (например, `-5:00`), либо использовать сочетание названия и сокращения часового пояса. Если указано только имя региона, то при наличии неоднозначности в определении времени (летнее или стандартное) Oracle будет считать, что задано стандартное время.

Полный список поддерживаемых Oracle названий регионов и часовых поясов содержит представление `V$TIMEZONE_NAMES`, доступное для всех пользователей базы данных. Анализируя информацию этого представления, обратите внимание на то, что сокращения часовых поясов не уникальны (см. следующую врезку).

О СТАНДАРТЕ ЧАСОВЫХ ПОЯСОВ

Часовые пояса настолько важны, что, казалось бы, должен существовать некий международный стандарт, определяющий их названия и сокращения. Тем не менее такого стандарта нет. Названия часовых поясов и сокращения не только не стандартизированы, но среди них даже встречаются повторения. Например, сокращение EST обозначает Восточное стандартное время и в США, и в Австралии, то есть соответствует двум часовым поясам с разным смещением времени. Сокращение BST применяется для нескольких часовых поясов, включая Тихоокеанский/Мидуэй и Лондонский, отличающиеся на 12 часов в режиме летнего времени или на 11 часов в остальное время года. Вот почему функция `TO_TIMESTAMP` не позволяет задавать часовой пояс лишь по сокращению названия.

Поскольку единого стандарта часовых поясов не существует, резонно спросить — откуда взяты имена регионов в представлении `V$TIMEZONE_NAMES`? Источником информации Oracle являются документы из каталога <ftp://elsie.nci.nih.gov/pub/>. Ищите файлы с именами вида `tzdataxxx.tar.gz`, где XXX — версия данных. Обычно в архиве хранится страница с именем `tz-link.htm`, содержащая дополнительную информацию и ссылки на URL-адреса других документов, относящихся к часовым поясам.

Точное совпадение маски форматирования

При преобразовании символьной строки в дату/время функции преобразования `TO_*` обычно руководствуются несколькими предположениями:

- Лишние пробелы в символьной строке игнорируются.
- Числовые значения (например, день года) не обязаны включать начальные нули для заполнения маски.
- Знаки препинания в преобразуемой строке могут просто совпадать со знаками препинания в маске по длине и позиции.

Такая гибкость очень удобна — до того момента, когда вы захотите ограничить пользователя или даже пакетный процесс от ввода данных в нестандартном формате. В некоторых ситуациях разделение дня и месяца символом «^» вместо дефиса («-») попросту непустимо. В таких ситуациях можно включить в маску форматирования модификатор `FX`, чтобы потребовать точного совпадения между строкой и маской форматирования.

С модификатором `FX` какая-либо гибкость в интерпретации строки полностью отсутствует. Строка не может содержать лишние пробелы, если их нет в модели. Ее числовые значения должны включать начальные нули, если модель форматирования включает дополнительные цифры. Наконец, знаки препинания и литералы должны точно соответствовать знакам препинания и заключенному в кавычки тексту маски форматирования (не считая регистра символов, который всегда игнорируется). Все эти правила продемонстрированы в следующих примерах:

```
TO_DATE ('1-1-4', 'fxDD-MM-YYYY')
TO_DATE ('7/16/94', 'FXMM/DD/YY')
TO_DATE ('JANUARY^1^ the year of 94', 'FXMonth-dd-"WhatIsaynotdo"yy')
```

PL/SQL выдает одну из следующих ошибок:

```
ORA-01861: literal does not match format string
ORA-01862: the numeric value does not match the length of the format item
```

Однако следующий пример выполняется успешно, потому что регистр символов не учитывается, и модификатор `FX` этого факта не меняет:

```
TO_DATE ('Jan 15 1994', 'fxMON DD YYYY')
```

Модификатор `FX` может задаваться в верхнем регистре, нижнем регистре или со смешением регистров; его действие от этого не изменяется.

Модификатор `FX` работает как переключатель и может многократно встречаться в маске форматирования. Пример:

```
TO_DATE ('07-1-1994', 'FXDD-FXMM-FXYYYY')
```

Каждый раз, когда модификатор `FX` встречается в маске форматирования, происходит переключение. В приведенном примере точное совпадение обязательно для дня и года, но не для месяца.

Ослабление требований к точности совпадения

Модификатор `FM` (Fill Mode, «режим заполнения») в маске форматирования при вызове `TO_DATE` или `TO_TIMESTAMP` используется для заполнения строки пробелами или нулями, чтобы обеспечить успешное прохождение проверки `FX`. Пример:

```
TO_DATE ('07-1-94', 'FXfmDD-FXMM-FXYYYY')
```

Преобразование проходит успешно, потому что модификатор `FM` дополняет год 94 нулями, и тот превращается в 0094 (хотя трудно представить, чтобы вам когда-нибудь понадобилось делать что-то подобное). День 1 дополняется одним нулем и превращается в 01. Модификатор `FM` работает в режиме переключателя, как и `FX`.

Казалось бы, такое использование FM противоречит самой цели FX. Зачем использовать оба модификатора? Например, модификатор FX может применяться для принудительного использования конкретных ограничителей, тогда как FM ослабляет требование о вводе начальных нулей.

Интерпретация года из двух цифр

Переход в новое тысячелетие породил интерес к хранению года в формате из четырех цифр, потому что разработчики внезапно осознали неоднозначность часто встречающегося формата с двумя цифрами. Например, к какому году относится дата 1-Jan-45 — к 1945 или 2045? В таких ситуациях лучше всего использовать однозначный год с четырьмя цифрами. Но несмотря на это понимание, старые привычки изменяются с трудом, а внесение изменений в существующие системы сопряжено с большими трудностями. Возможно, ваши пользователи предпочитают вводить год из двух цифр, а не из четырех. Для подобных случаев Oracle предоставляет форматный элемент RR, интерпретирующий год из двух цифр в скользящем окне.



В последующем обсуждении термин «век» используется в его житейском понимании. К 20-му веку относятся годы 1900–1999, а к 21-му — годы 2000–2099. Я понимаю, что такое определение не совсем корректно, но оно упрощает объяснение поведения RR.

Если текущий год относится к первой половине века (годы с 0 по 49), то:

- при вводе даты, относящейся к первой половине века (то есть от 0 до 49) RR возвращает текущий век;
- при вводе даты, относящейся ко второй половине века (то есть от 50 до 99), RR возвращает предыдущий век.

Если же текущий год относится ко второй половине века (годы с 50 по 99), то:

- при вводе даты, относящейся к первой половине века, RR возвращает следующий век;
- при вводе даты, относящейся ко второй половине века, RR возвращает текущий век.

Запутались? Я тоже разобрался не сразу. Правила RR пытаются предположить, какой век подразумевал пользователь, если он не был указан явно.

Рассмотрим несколько примеров воздействия RR. Обратите внимание: для годов 88 и 18 SYSDATE возвращает текущую дату из 20 и 21 века соответственно:

```
SQL> SELECT TO_CHAR (SYSDATE, 'MM/DD/YYYY') "Current Date",
2          TO_CHAR (TO_DATE ('14-OCT-88', 'DD-MON-RR'), 'YYYY') "Year 88",
3          TO_CHAR (TO_DATE ('14-OCT-18', 'DD-MON-RR'), 'YYYY') "Year 18"
FROM dual;
```

```
Current Date Year 88 Year 18
-----
02/25/2014    1988    2018
```

Когда мы достигаем года 2050, RR интерпретирует те же даты по-другому:

```
SQL> SELECT TO_CHAR (SYSDATE, 'MM/DD/YYYY') "Current Date",
2          TO_CHAR (TO_DATE ('10/14/88', 'MM/DD/RR'), 'YYYY') "Year 88",
3          TO_CHAR (TO_DATE ('10/14/18', 'MM/DD/RR'), 'YYYY') "Year 18"
4          FROM dual;
```

```
Current Date Year 88 Year 18
-----
02/25/2050    2088    2118
```

Логику RR в текущих приложениях можно активизировать несколькими способами. Самый простой и элегантный способ — изменение маски форматирования дат по умолчанию в экземпляре базы данных. Собственно, Oracle уже делает это за нас. В стандартной установке Oracle параметр NLS_DATE_FORMAT задается следующим образом:

```
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-RR';
```

Если маска форматирования дат не будет жестко запрограммирована на других экранах или отчетах, год из двух цифр будет интерпретироваться по правилам, приведенным выше.

Преобразование часовых поясов в символьные строки

Часовые пояса усложняют преобразование значений даты/времени в символьные строки. Информация часового пояса состоит из следующих элементов.

- Смещение относительно UTC в часах и минутах.
- Имя региона часового пояса.
- Сокращение часового пояса.

Все эти элементы хранятся отдельно в переменной **TIMESTAMP WITH TIME ZONE**. Смещение относительно UTC присутствует всегда, но возможность вывода названия региона или сокращения зависит от того, задана эта информация или нет. Рассмотрим следующий пример:

```
DECLARE
  ts1 TIMESTAMP WITH TIME ZONE;
  ts2 TIMESTAMP WITH TIME ZONE;
  ts3 TIMESTAMP WITH TIME ZONE;
BEGIN
  ts1 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 5:00',
    'YYYY-MM-DD HH24:MI:SS.FF TZH:TZM');
  ts2 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 US/Eastern',
    'YYYY-MM-DD HH24:MI:SS.FF TZR');
  ts3 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 US/Eastern EDT',
    'YYYY-MM-DD HH24:MI:SS.FF TZR TZD');

  DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts1,
    'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts2,
    'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts3,
    'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
END;
```

Результат:

```
2002-06-18 01:52:00.123457000 PM -05:00 -05:00
2002-06-18 01:52:00.123457000 PM -04:00 US/EASTERN EDT
2002-06-18 01:52:00.123457000 PM -04:00 US/EASTERN EDT
```

В этом коде следует обратить внимание на некоторые аспекты, относящиеся к информации часовых поясов:

- В значении, присвоенном переменной **ts1**, часовой пояс задается смещением относительно UTC. Соответственно при выводе **ts1** выводится только смещение.
- При отсутствии имени региона для **ts1** Oracle предоставляет смещение часового пояса. Это все же лучше, чем полное отсутствие информации.
- В значении, присвоенном переменной **ts2**, указывается регион часового пояса. Внутреннее преобразование переводит его в смещение относительно UTC, но имя региона при этом сохраняется. Таким образом, для **ts2** может быть выведено как смещение UTC, так и имя региона.

- Для переменной `ts2` Oracle правильно распознает действие летнего времени в июне. В результате значение `ts2` неявно связывается с сокращением EDT.
- В значении, присвоенном переменной `ts3`, задается как регион, так и сокращение часового пояса; соответственно, оба значения могут быть выведены.

Смещения относительно UTC и регионы часовых поясов связаны отношениями «один ко многим»; смещения недостаточны для однозначного определения имени региона. Вот почему невозможно вывести имя региона, если оно не было задано изначально.

Дополнение вывода с модификатором FM

Модификатор FM, описанный в разделе «Ослабление требований к точности совпадения», также может использоваться для преобразования даты/времени в символьную строку для подавления дополняющих пробелов и начальных нулей, которые могут быть возвращены функцией `TO_CHAR`.

По умолчанию следующая маска форматирования генерирует как дополняющие пробелы, так и начальные нули (название месяца отделяется от дня пятью пробелами):

```
TO_CHAR (SYSDATE, 'Month DD, YYYY') --> 'April      05, 1994'
```

Однако с модификатором FM в начале маски и лишний пробел, и начальные нули исчезают:

```
TO_CHAR (SYSDATE, 'FMMonth DD, YYYY') --> April 5, 1994'
```

Модификатор может задаваться в верхнем регистре, нижнем регистре или со смещением регистров; его действие от этого не изменяется.

Помните, что модификатор FM работает как переключатель и может многократно встречаться в маске форматирования. Каждый раз, когда он встречается в маске форматирования, происходит переключение. По умолчанию (то есть если модификатор FM вообще не встречается в маске) пробелы не подавляются, а начальные нули включаются в итоговое значение.

Литералы типа DATE и TIMESTAMP

Литералы DATE и TIMESTAMP (а также интервальные литералы, которые будут описаны позднее в этой главе) являются частью стандарта ISO SQL и поддерживаются, начиная с Oracle9i. Они представляют еще одну возможность записи значений в переменные типа даты/времени. *Литерал даты* состоит из ключевого слова DATE, за которым следует дата (и только дата!) в следующем формате:

```
DATE 'YYYY-MM-DD'
```

Литерал TIMESTAMP состоит из ключевого слова TIMESTAMP, за которым указывается дата и время в строго определенном формате:

```
TIMESTAMP 'YYYY-MM-DD HH:MI:SS[.FFFFFFFFF] [{+|-}HH:MI]'
```

Обозначение FFFFFFFFFF представляет доли секунды и не является обязательным. Доли секунды занимают от одной до девяти цифр. Необязательное смещение часового пояса (+HH:MI) может содержать знак «плюс» или «минус». Часы всегда задаются в 24-часовом формате.



Если в литерале типа TIMESTAMP не указано смещение часового пояса, по умолчанию будет использоваться часовой пояс текущего сеанса.

В следующем коде PL/SQL представлено несколько допустимых литералов типа `DATE` и `TIMESTAMP`:

```
DECLARE
    ts1 TIMESTAMP WITH TIME ZONE;
    ts2 TIMESTAMP WITH TIME ZONE;
    ts3 TIMESTAMP WITH TIME ZONE;
    ts4 TIMESTAMP WITH TIME ZONE;
    ts5 DATE;
BEGIN
    --Две цифры для долей секунды
    ts1 := TIMESTAMP '2002-02-19 11:52:00.00 -05:00';

    --9 цифр для долей секунды, 24-часовой формат.
    ts2 := TIMESTAMP '2002-02-19 14:00:00.000000000 -5:00';

    --Без долей секунды
    ts3 := TIMESTAMP '2002-02-19 13:52:00 -5:00';

    --Без часового пояса, по умолчанию используется часовой пояс текущего сеанса
    ts4 := TIMESTAMP '2002-02-19 13:52:00';

    --Литерал типа DATE
    ts5 := DATE '2002-02-19';
END;
```

Формат литералов `DATE` и `TIMESTAMP` определяется стандартами ANSI/ISO; изменить его не сможете ни вы, ни администратор базы данных. Поэтому использование этих литералов уместно везде, где в программном коде должны задаваться конкретные значения (например, константы) даты/времени.



Oracle позволяет использовать в литералах `TIMESTAMP` имена регионов часовых поясов — например, `TIMESTAMP '2002-02-19 13:52:00 EST'`. Однако следует помнить, что эта функциональность выходит за рамки стандарта SQL.

Таблица 10.2. Имена интервальных элементов

| Имя | Описание |
|--------|--|
| YEAR | Количество лет в диапазоне от 1 до 999 999 999 |
| MONTH | Количество месяцев в диапазоне от 0 до 11 |
| DAY | Количество дней в диапазоне от 0 до 999 999 999 |
| HOURL | Количество часов в диапазоне от 0 до 23 |
| MINUTE | Количество минут в диапазоне от 0 до 59 |
| SECOND | Количество секунд в диапазоне от 0 до 59,999 999 999 |

Преобразования интервалов

Интервал состоит из одного или нескольких элементов даты/времени. Например, интервал можно выразить в годах и месяцах, часах и минутах и т. д. В табл. 10.2 перечислены стандартные имена всех этих элементов. Они используются в функциях преобразования типов данных и в выражениях, о чем подробно рассказывается в следующих разделах. При использовании в функциях преобразования типов имена элементов интервала не чувствительны к регистру. Например, обозначения `YEAR`, `Year` и `year` эквивалентны.

Преобразование чисел в интервалы

Функции `NUMTOYMINTERVAL` и `NUMTODSINTERVAL` предназначены для преобразования отдельных числовых значений в значения типа `INTERVAL`. При вызове функции задаются числовое значение и один из элементов интервала, перечисленных в табл. 10.2.

Функция `NUMTOYMINTERVAL` преобразует числовое значение в интервал типа `INTERVAL YEAR TO MONTH`, а функция `NUMTODSINTERVAL` — в интервал типа `INTERVAL DAY TO SECOND`.

В следующем примере функция `NUMTOYMINTERVAL` используется для преобразования значения 10.5 в значение типа `INTERVAL YEAR TO MONTH`. Вторым аргументом — строка `Year` — указывает, что заданное в первом аргументе число представляет количество лет:

```
DECLARE
  y2m INTERVAL YEAR TO MONTH;
BEGIN
  y2m := NUMTOYMINTERVAL (10.5, 'Year');
  DBMS_OUTPUT.PUT_LINE(y2m);
END;
```

Результат выполнения:

```
+10-06
```

Как видите, исходное значение 10.5 преобразовано в значение, соответствующее 10 годам и 6 месяцам. Любое дробное количество лет преобразуется в эквивалентное количество месяцев, и результат округляется до целого. Например, если исходное значение равно 10,9 года, оно будет преобразовано в 10 лет и 10 месяцев. В следующем примере числовое значение преобразуется в интервал типа `INTERVAL DAY TO SECOND`:

```
DECLARE
  an_interval INTERVAL DAY TO SECOND;
BEGIN
  an_interval := NUMTODSINTERVAL (1440, 'Minute');
  DBMS_OUTPUT.PUT_LINE(an_interval);
END;
```

Результат:

```
+01 00:00:00.000000
```

PL/SQL procedure successfully completed.

Oracle автоматически преобразует входное значение 1400 минут в интервал, равный одному дню. Это удобно, потому что вам не придется заниматься этой работой самостоятельно. С помощью функций `NUMTO` вы сможете легко вывести любое количество минут (секунд, дней или часов) в нормализованном формате, понятном для читателя. До появления интервальных типов вам пришлось бы самостоятельно программировать преобразование минут в правильное количество дней, часов и минут.

Преобразование строк в интервалы

Если функции `NUMTO` предназначены для преобразования в интервалы числовых значений, то для преобразования символьных строк можно использовать функции `TO_YMINTERVAL` и `TO_DSINTERVAL`. Выбор функции зависит от того, значение какого типа вы хотите получить — `INTERVAL YEAR TO MONTH` или `INTERVAL DAY TO SECOND`.

Функция `TO_YMINTERVAL` преобразует символьную строку в значение типа `INTERVAL YEAR TO MONTH`. Вызывается она так:

```
TO_YMINTERVAL('Y-M')
```

Здесь аргумент *Y* представляет количество лет, а *M* — количество месяцев. Обязательно должны быть заданы оба значения, разделенные дефисом.

Аналогичным образом функция `TO_DSINTERVAL` преобразует символьную строку в значение типа `INTERVAL DAY TO SECOND`. А вот как она вызывается:

```
TO_DSINTERVAL('D HH:MI:SS.FF')
```

Здесь *D* — количество дней, а аргумент *HH:MM:SS.FF* представляет часы, минуты, секунды и доли секунд.

Следующий пример демонстрирует использование обеих функций:

```
DECLARE
  y2m INTERVAL YEAR TO MONTH;
  d2s1 INTERVAL DAY TO SECOND;
  d2s2 INTERVAL DAY TO SECOND;
BEGIN
  y2m := TO_YMINTERVAL('40-3'); --мой возраст
  d2s1 := TO_DSINTERVAL('10 1:02:10');
  d2s2 := TO_DSINTERVAL('10 1:02:10.123'); --доли секунды
END;
```

При вызове любой из них необходимо задавать полный набор значений. Например, нельзя вызвать функцию `TO_YMINTERVAL`, указав только год, или вызвать функцию `TO_DS_INTERVAL` без секунд. Можно опустить только доли секунд.

Форматирование интервалов для вывода

До сих пор речь шла о преобразовании интервалов и выводе данных с использованием механизма неявного преобразования значений. К сожалению, этим все и ограничивается. Интервал можно передать при вызове `TO_CHAR`, но функция игнорирует маску форматирования. Например, выполнение кода

```
DECLARE
  y2m INTERVAL YEAR TO MONTH;
BEGIN
  y2m := INTERVAL '40-3' YEAR TO MONTH;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(y2m, 'YY "Years" and MM "Months"'));
END;
```

приведет к такому результату, как если бы маска отсутствовала:

```
+000040-03
```

Если вас не устраивает то, как выполняется неявное преобразование значения интервала в символьную строку, воспользуйтесь функцией `EXTRACT`:

```
DECLARE
  y2m INTERVAL YEAR TO MONTH;
BEGIN
  y2m := INTERVAL '40-3' YEAR TO MONTH;

  DBMS_OUTPUT.PUT_LINE(
    EXTRACT(YEAR FROM y2m) || ' лет и '
    || EXTRACT(MONTH FROM y2m) || ' месяца'
  );
END;
```

Теперь результат будет таким:

```
40 лет и 3 месяца
```

Функция `EXTRACT` более подробно описана далее в разделе «CAST и EXTRACT».

Литералы типа INTERVAL

Литералы типа INTERVAL сходны с литералами TIMESTAMP; они используются для вставки интервальных значений в виде констант в программный код. Литералы типа INTERVAL имеют следующий синтаксис:

INTERVAL '*символьное_представление*' *начальный_элемент* TO *конечный_элемент*

Здесь *символьное_представление* — символьная строка, представляющая интервал (описание представления двух типов интервалов в символьной форме приводится выше, в разделе «Преобразования интервалов»); *начальный_элемент* задает начальный элемент интервала, а *конечный_элемент* задает конечный элемент интервала.

В отличие от функций TO_YMINTERVAL и TO_DSINTERVAL, литералы INTERVAL позволяют задать интервал с использованием любой последовательности элементов даты/времени (см. табл. 10.2). Существует только два ограничения: элементы должны быть последовательными, и в одном интервале не допускается переход от месяцев к дням.

Приведем несколько примеров:

```
DECLARE
  y2ma INTERVAL YEAR TO MONTH;
  y2mb INTERVAL YEAR TO MONTH;
  d2sa INTERVAL DAY TO SECOND;
  d2sb INTERVAL DAY TO SECOND;
BEGIN
  /* Несколько примеров интервалов YEAR TO MONTH */
  y2ma := INTERVAL '40-3' YEAR TO MONTH;
  y2mb := INTERVAL '40' YEAR;

  /* Примеры интервалов DAY TO SECOND */
  d2sa := INTERVAL '10 1:02:10.123' DAY TO SECOND;

  /* Не работает в Oracle9i - 11gR2 из-за ошибки */
  --d2sb := INTERVAL '1:02' HOUR TO MINUTE;

  /* Два обходных способа задания интервала HOUR TO MINUTE,
     представляющего только часть интервала DAY TO SECOND range. */
  SELECT INTERVAL '1:02' HOUR TO MINUTE
  INTO d2sb
  FROM dual;

  d2sb := INTERVAL '1' HOUR + INTERVAL '02' MINUTE;
END;
```



Начиная с Oracle9i и вплоть до Oracle11g Release 2, выражения вида INTERVAL '1:02' HOUR TO MINUTE, в которых не указывается значение каждого возможного элемента, могут использоваться в командах SQL, но не в командах PL/SQL. Более того, при выполнении программы будет выдано сообщение об использовании ключевого слова BULK в неверном контексте. По всей вероятности, эта ошибка будет исправлена в следующих выпусках продукта.

Oracle нормализует значения интервалов, что весьма удобно. В следующем примере 72 часа 15 минут преобразуются в 3 дня, 0 часов и 15 минут:

```
DECLARE
  d2s INTERVAL DAY TO SECOND;
BEGIN
  SELECT INTERVAL '72:15' HOUR TO MINUTE INTO d2s FROM DUAL;
  DBMS_OUTPUT.PUT_LINE(d2s);
END;
```

Результат:

+03 00:15:00.000000

Oracle почему-то нормализует только старшие значения (часы в данном примере). Попробка задать интервал 72:75 (72 часа и 75 минут) приведет к ошибке.

CAST и EXTRACT

Стандартные функции SQL CAST и EXTRACT часто бывают полезными при работе с датой/временем. Функция CAST появилась в Oracle8 как механизм явного определения типов коллекций, а в Oracle8i ее возможности были расширены. Теперь функция CAST может использоваться для преобразования значений даты/времени в символьные строки, и наоборот. Функция EXTRACT, введенная в Oracle9i, позволяет выделять отдельные компоненты из значений даты/времени и интервалов.

Функция CAST

Функция CAST может использоваться для выполнения следующих операций с датой/временем:

- преобразование символьной строки в значение даты/времени;
- преобразование значения даты/времени в символьную строку;
- преобразование значения одного типа даты/времени (например, DATE) в значение другого типа (например, TIMESTAMP).

При использовании функции CAST для преобразования значений даты/времени в символьные строки и обратно учитываются значения параметров NLS. Чтобы проверить свои настройки NLS, запросите представление V\$NLS_PARAMETERS, а для их изменения используется команда ALTER SESSION. Параметры NLS для даты/времени:

- NLS_DATE_FORMAT — используется при преобразовании в тип данных DATE и обратно;
- NLS_TIMESTAMP_FORMAT — используется при преобразовании в типы данных TIMESTAMP и TIMESTAMP WITH LOCAL TIME ZONE.
- NLS_TIMESTAMP_TZ_FORMAT — используется при преобразовании в тип данных TIMESTAMP WITH TIME ZONE и обратно.

В следующем примере показано, как выполняется каждый тип преобразования. Предполагается, что для NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT и NLS_TIMESTAMP_TZ_FORMAT используются значения по умолчанию 'DD-MON-RR', 'DD-MON-RR HH.MI.SSXFF AM' и 'DD-MON-RR HH.MI.SSXFF AM TZR' соответственно.

```
DECLARE
    tstz TIMESTAMP WITH TIME ZONE;
    string VARCHAR2(40);
    tsltz TIMESTAMP WITH LOCAL TIME ZONE;
BEGIN
    -- Преобразование строки в дату/время
    tstz := CAST ('24-Feb-2009 09.00.00.00 PM US/Eastern'
                  AS TIMESTAMP WITH TIME ZONE);
    -- Преобразование даты/времени обратно в строку
    string := CAST (tstz AS VARCHAR2);
    tsltz := CAST ('24-Feb-2009 09.00.00.00 PM'
                  AS TIMESTAMP WITH LOCAL TIME ZONE);

    DBMS_OUTPUT.PUT_LINE(tstz);
    DBMS_OUTPUT.PUT_LINE(string);
    DBMS_OUTPUT.PUT_LINE(tsltz);
END;
```

Результат:

```
24-FEB-09 09.00.00.000000 PM US/EASTERN
24-FEB-09 09.00.00.000000 PM US/EASTERN
24-FEB-09 09.00.00.000000 PM
```

Здесь на основе символьной строки генерируется значение типа `TIMESTAMP WITH TIME ZONE`, которое преобразуется в `VARCHAR2`, а затем в `TIMESTAMP WITH LOCAL TIME ZONE`.

Резонно спросить — зачем использовать `CAST`? Действительно, эта функция частично перекрывается с функциями `TO_DATE`, `TO_TIMESTAMP` и `TO_TIMESTAMP_TZ`. Однако функция `TO_TIMESTAMP` может получать в качестве входных данных только строку, тогда как `CAST` может получить строку или `DATE` и преобразовать полученное значение в `TIMESTAMP`. Таким образом, `CAST` используется в тех случаях, когда возможностей функций `TO_` оказывается недостаточно. Но если задачу можно решить при помощи функции `TO_`, лучше использовать именно ее, потому что код обычно получается более понятным.



В команде SQL при вызове `CAST` можно указать размер значения типа данных, например `CAST (x AS VARCHAR2(40))`. Однако PL/SQL не позволяет задать размер значения целевого типа данных.

Функция EXTRACT

Функция `EXTRACT` используется для извлечения компонентов из значения даты/времени. Синтаксис:

```
EXTRACT (имя_компонента, FROM {дата_время | интервал})
```

Здесь *имя_компонента* — имя одного из элементов даты/времени, перечисленных в табл. 10.3. Имена компонентов не чувствительны к регистру символов. В аргументе *дата_время* или *интервал* задается действительное значение даты/времени или интервала. Тип возвращаемого функцией значения зависит от извлекаемого компонента.

Таблица 10.3. Имена компонентов даты/времени для функции `EXTRACT`

| Имя компонента | Возвращаемый тип данных |
|-----------------|-------------------------|
| YEAR | NUMBER |
| MONTH | NUMBER |
| DAY | NUMBER |
| HOUR | NUMBER |
| MINUTE | NUMBER |
| SECOND | NUMBER |
| TIMEZONE_HOUR | NUMBER |
| TIMEZONE_MINUTE | NUMBER |
| TIMEZONE_REGION | VARCHAR2 |
| TIMEZONE_ABBR | VARCHAR2 |

В следующем примере функция `EXTRACT` используется для проверки того, является ли текущий месяц ноябрем:

```
BEGIN
  IF EXTRACT (MONTH FROM SYSDATE) = 11 THEN
    DBMS_OUTPUT.PUT_LINE('Сейчас ноябрь');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Сейчас не ноябрь');
  END IF;
END;
```

Функцию `EXTRACT` удобно использовать в тех случаях, когда управление потоком выполнения программы осуществляется в зависимости от значения одного из элементов даты/времени, а также в тех случаях, когда требуется получить числовое значение одного из элементов даты/времени.

Арифметические операции над значениями даты/времени

Основные операции над значениями даты/времени в Oracle сводятся к следующему набору:

- Прибавление или вычитание интервала из значения даты/времени.
- Вычитание одного значения даты/времени из другого для определения интервала между двумя значениями.
- Прибавление или вычитание одного интервала из другого.
- Умножение или деление интервала на число.

По историческим причинам я раздельно рассматриваю арифметические операции со значениями типа `DATE` и операции, в которых задействованы типы семейств `TIMESTAMP` и `INTERVAL`.

Операции с типами `TIMESTAMP` и `INTERVAL`

Вычисления с интервалами «дни/секунды» легко выполняются при работе с типами данных семейства `TIMESTAMP`. Создайте значение `INTERVAL DAY TO SECOND` и используйте его при сложении и вычитании. Например, прибавление к текущей дате 1500 дней, 4 часов, 30 минут и 2 секунд выполняется следующим образом:

```
DECLARE
    current_date TIMESTAMP;
    result_date TIMESTAMP;
BEGIN
    current_date := SYSTIMESTAMP;
    result_date := current_date + INTERVAL '1500 4:30:2' DAY TO SECOND;
    DBMS_OUTPUT.PUT_LINE(result_date);
END;
```

С интервалами «годы/месяцы» дело обстоит сложнее. Продолжительность любого дня составляет 24 часа, или 1440 минут, или даже 86 400 секунд, но не все месяцы имеют одинаковую продолжительность в днях: 28, 29, 30 или 31 день. По этой причине простое прибавление одного месяца к дате может привести к неоднозначному результату. Допустим, вы прибавляете один месяц к последнему дню мая; что получится — последний день июня или недействительная дата 31 июня? Все зависит от того, что должны представлять интервалы.

Oracle предоставляет необходимые средства для получения любого из этих результатов. Программист сам решает, какой вариант поведения должен реализоваться системой. Если конец месяца должен быть преобразован в конец месяца (31 мая + 1 месяц = 30 июня), используйте функцию `ADD_MONTHS`. Если изменение дня месяца нежелательно, используйте значение `INTERVAL YEAR TO MONTH`. В этом случае при прибавлении к 31 мая 2008 года `INTERVAL '1' MONTH` будет получено значение 31 июня 2008 года, а СУБД выдаст сообщение об ошибке.

Вычисления с типом `INTERVAL YEAR TO MONTH` лучше зарезервировать для тех значений даты/времени, которые усекаются по началу месяца или, скажем, по 15 числу — они плохо подходят для конца месяца. Если вам потребуется прибавить или вычесть сколько-то

месяцев (или лет — аналогичная проблема возникает при прибавлении одного кода к 29 февраля 2008 года) из даты, в которой может быть задействован конец месяца, используйте функцию `ADD_MONTHS`. Эта функция, возвращающая тип `DATE`, решает проблему преобразованием соответствующих дат в последний день месяца вместо выдачи ошибки. Например, `ADD_MONTHS('31-May-2008', 1)` вернет 30 июня 2008 года. Полученное значение `DATE` не содержит данных часового пояса (или долей секунд); если в результате должны присутствовать эти компоненты, вам придется реализовать дополнительную логику извлечения и повторного внесения этих компонентов в результат.

```
DECLARE
    end_of_may2008 TIMESTAMP;
    next_month TIMESTAMP;
BEGIN
    end_of_may2008 := TO_TIMESTAMP('31-May-2008', 'DD-Mon-YYYY');
    next_month := TO_TIMESTAMP(ADD_MONTHS(end_of_may2008, 1));
    DBMS_OUTPUT.PUT_LINE(next_month);
END;
```

Результат:

30-Jun-2008 00:00:00.000000

Аналогичной функции вычитания `SUBTRACT_MONTHS` не существует, но `ADD_MONTHS` можно вызывать с отрицательным количеством месяцев. Например, вызов `ADD_MONTHS(current_date, -1)` в приведенном примере вернет дату за один месяц до последнего дня апреля.

Операции с типом DATE

В операциях с типом `DATE` можно использовать как значения `INTERVAL`, так и числовые значения, представляющие дни и их доли. Например, прибавление одного дня к текущей дате и времени выполняется так:

```
SYSDATE + 1
```

Прибавление четырех часов к текущей дате и времени:

```
SYSDATE + (4/24)
```

Обратите внимание на использование дроби $4/24$ вместо $1/6$. При чтении кода сразу становится ясно, что значение, возвращаемое `SYSDATE`, увеличивается на 4 часа; а если использовать $1/6$, программист, который будет заниматься сопровождением кода, будет долго ломать голову над тем, что должна означать эта таинственная дробь. Для еще более явного выражения намерений можно воспользоваться именованной константой:

```
DECLARE
    four_hours NUMBER := 4/24;
BEGIN
    DBMS_OUTPUT.PUT_LINE(
        'Now + 4 hours = ' || TO_CHAR(SYSDATE + four_hours));
END;
```

В табл. 10.4 приведены дробные значения, представляющие часы, минуты и секунды при работе с `DATE`. Также в нее включены некоторые дробные значения, которые могут использоваться для построения этих значений.

Таблица 10.4. Дробные значения, используемые для представления даты в арифметических операциях

| Значение | Выражение | Представляет |
|----------|------------|--------------|
| 1/24 | 1/24 | Один час |
| 1/1440 | 1/24/60 | Одна минута |
| 1/86400 | 1/24/60/60 | Одна секунда |

Вычисление интервала между двумя значениями DATE

```
DECLARE
    leave_on_trip TIMESTAMP := TIMESTAMP '2005-03-22 06:11:00.00';
    return_from_trip TIMESTAMP := TIMESTAMP '2005-03-25 15:50:00.00';
    trip_length INTERVAL DAY TO SECOND;
BEGIN
    trip_length := return_from_trip - leave_on_trip;

    DBMS_OUTPUT.PUT_LINE('Длина в формате дни часы:минуты:секунды
    ' || trip_length);
END;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE (
    TO_DATE('25-Mar-2005 3:50 pm','dd-Mon-yyyy hh:mi am')
    - TO_DATE('22-Mar-2005 6:11 am','dd-Mon-yyyy hh:mi am')
  );
END;
```

- Если *дата_1* наступает позже *даты_2*, MONTHS_BETWEEN возвращает положительное число.

- Если *дата_1* наступает раньше *даты_2*, MONTHS_BETWEEN возвращает отрицательное число.
- Если *дата_1* и *дата_2* относятся к одному месяцу, функция возвращает дробное значение из диапазона от -1 до $+1$.
- Если *дата_1* и *дата_2* приходятся на последние дни соответствующих месяцев, функция возвращает целое число (без дробного компонента).
- Если *дата_1* и *дата_2* относятся к разным месяцам и хотя бы одна из них не приходится на последний день месяца, функция возвращает дробное значение. (Дробный компонент вычисляется для месяцев, состоящих из 31 дня, с учетом разницы компонентов времени двух дат.)

Несколько примеров использования MONTHS_BETWEEN:

BEGIN

```
-- Разность между двумя последними днями месяцев,
-- первый месяц наступает раньше второго:
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('31-JAN-1994', '28-FEB-1994'));

-- Разность между двумя последними днями месяцев,
-- первый месяц наступает позже второго:
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('31-MAR-1995', '28-FEB-1994'));

-- Две даты одного месяца:
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('28-FEB-1994', '15-FEB-1994'));

-- Вычисления с дробным компонентом:
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('31-JAN-1994', '1-MAR-1994'));
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('31-JAN-1994', '2-MAR-1994'));
DBMS_OUTPUT.PUT_LINE(
    MONTHS_BETWEEN ('31-JAN-1994', '10-MAR-1994'));
```

END;

Результаты:

```
-1
13
.4193548387096774193548387096774193548387
-1.03225806451612903225806451612903225806
-1.06451612903225806451612903225806451613
-1.32258064516129032258064516129032258065
```

Вероятно, вы заметили здесь определенную закономерность. Как уже было сказано, функция MONTHS_BETWEEN вычисляет дробный компонент количества месяцев исходя из предположения, что каждый месяц содержит 31 день. Поэтому на каждый день сверх полного месяца к результату прибавляется $1/31$ месяца:

$1/31 = .032258065...$

В соответствии с этим правилом количество месяцев между 31 января 1994 года и 28 февраля 1994 года равно 1 — удобное целое число. Однако количество месяцев между 31 января 1994 года и 1 марта 1994 года увеличивается на .032258065. Как и в случае с вычитанием DATE, при работе с MONTHS_BETWEEN часто используется функция TRUNC.

Смешанное использование DATE и TIMESTAMP

Результатом вычитания двух TIMESTAMP является значение типа INTERVAL DAY TO SECOND. Результат вычитания с двумя значениями DATE представляет собой числовое значение.

Соответственно, если требуется вычесть одно значение DATE из другого и вернуть значение INTERVAL DAY TO SECOND, вам придется преобразовать DATE в TIMESTAMP функцией CAST. Пример:

```
DECLARE
    dt1 DATE;
    dt2 DATE;
    d2s INTERVAL DAY(3) TO SECOND(0);
BEGIN
    dt1 := TO_DATE('15-Nov-1961 12:01 am', 'dd-Mon-yyyy hh:mi am');
    dt2 := TO_DATE('18-Jun-1961 11:59 pm', 'dd-Mon-yyyy hh:mi am');

    d2s := CAST(dt1 AS TIMESTAMP) - CAST(dt2 AS TIMESTAMP);

    DBMS_OUTPUT.PUT_LINE(d2s);
END;
```

Результат:

```
+149 00:02:00
```

Если значения DATE и TIMESTAMP смешиваются в одном выражении вычитания, PL/SQL выполняет неявное преобразование DATE в TIMESTAMP. Пример:

```
DECLARE
    dt DATE;
    ts TIMESTAMP;
    d2s1 INTERVAL DAY(3) TO SECOND(0);
    d2s2 INTERVAL DAY(3) TO SECOND(0);
BEGIN
    dt := TO_DATE('15-Nov-1961 12:01 am', 'dd-Mon-yyyy hh:mi am');
    ts := TO_TIMESTAMP('18-Jun-1961 11:59 pm', 'dd-Mon-yyyy hh:mi am');
    d2s1 := dt - ts;
    d2s2 := ts - dt;
    DBMS_OUTPUT.PUT_LINE(d2s1);
    DBMS_OUTPUT.PUT_LINE(d2s2);
END;
```

Результат:

```
+149 00:02:00
-149 00:02:00
```

Как обычно при работе с типами данных даты и времени, в программе желательно использовать явные преобразования.

Сложение и вычитание интервалов

В отличие от значений даты/времени, операция суммирования интервалов выглядит вполне разумно. Также имеет смысл и вычитание одного интервала из другого. Необходимо лишь помнить, что интервалы, участвующие в суммировании или вычитании, должны относиться к одному типу. Например:

```
DECLARE
    dts1 INTERVAL DAY TO SECOND := '2 3:4:5.6';
    dts2 INTERVAL DAY TO SECOND := '1 1:1:1.1';

    ytm1 INTERVAL YEAR TO MONTH := '2-10';
    ytm2 INTERVAL YEAR TO MONTH := '1-1';

    days1 NUMBER := 3;
    days2 NUMBER := 1;
BEGIN
    DBMS_OUTPUT.PUT_LINE(dts1 - dts2);
    DBMS_OUTPUT.PUT_LINE(ytm1 - ytm2);
    DBMS_OUTPUT.PUT_LINE(days1 - days2);
END;
```


Результат:

```
+0000000001 02:03:04.500000000
+0000000001-09
2
```

Пример демонстрирует результаты трех вычитаний интервалов. В первых двух операциях участвуют интервалы `INTERVAL DAY TO SECOND` и `INTERVAL YEAR TO MONTH`. В третьей операции используется вычитание двух чисел. Запомните: при работе с типами `DATE` интервал между двумя значениями `DATE` выражается типом `NUMBER`. Так как месяц может состоять из 28, 29, 30 или 31 дня, при попытке суммирования или вычитания интервала «дни/секунды» с интервалом «годы/месяцы» происходит ошибка.

Умножение и деление интервалов

Операции умножения и деления не применимы к датам, но зато интервал можно умножить или разделить на число. Несколько примеров:

```
DECLARE
  dts1 INTERVAL DAY TO SECOND := '2 3:4:5.6';
  dts2 INTERVAL YEAR TO MONTH := '2-10';
  dts3 NUMBER := 3;
BEGIN
  -- Умножение интервала
  DBMS_OUTPUT.PUT_LINE(dts1 * 2);
  DBMS_OUTPUT.PUT_LINE(dts2 * 2);
  DBMS_OUTPUT.PUT_LINE(dts3 * 2);

  -- Деление интервала
  DBMS_OUTPUT.PUT_LINE(dts1 / 2);
  DBMS_OUTPUT.PUT_LINE(dts2 / 2);
  DBMS_OUTPUT.PUT_LINE(dts3 / 2);
END;
```

Результат:

```
+0000000004 06:08:11.200000000
+0000000005-08
6
+0000000001 01:32:02.800000000
+0000000001-05
1.5
```

Типы данных INTERVAL без ограничений

Интервалы можно объявлять с разным уровнем точности, причем значения разной точностью не полностью совместимы между собой. Проблема особенно наглядно проявляется при написании процедур и функций, получающих параметры типа `INTERVAL`. Обратите внимание на потерю точности в следующем примере, где значение переменной `dts` удваивается с помощью функции `double_my_interval`:

```
DECLARE
  dts INTERVAL DAY(9) TO SECOND(9);
  FUNCTION double_my_interval (
    dts_in IN INTERVAL DAY TO SECOND) RETURN INTERVAL DAY TO SECOND
  IS
  BEGIN
    RETURN dts_in * 2;
  END;
BEGIN
  dts := '1 0:0:0.123456789';
  DBMS_OUTPUT.PUT_LINE(dts);
  DBMS_OUTPUT.PUT_LINE(double_my_interval(dts));
END;
```

Результат выполнения кода:

```
+0000000001 00:00:00.123456789
+02 00:00:00.246914
```

Цифры были потеряны не только в дробной части секунд, но и в значении количества дней. А если бы переменной `mts` было присвоено значение, равное 100 или более дням, попытка вызова функции `double_my_interval` привела бы к ошибке!

Дело в том, что задаваемая по умолчанию точность типов данных `INTERVAL` не равна максимально возможной точности. Обычно вызывающая программа передает точность параметров программе PL/SQL, но с типами данных `INTERVAL` используется принятая по умолчанию точность 2. Для решения этой проблемы можно воспользоваться типами данных `INTERVAL`, явно объявляемыми без ограничения точности:

- `YMINTERVAL_UNCONSTRAINED` — принимает любое значение типа `INTERVAL YEAR TO MONTH` без потери точности;
- `DSINTERVAL_UNCONSTRAINED` — принимает любое значение типа `INTERVAL DAY TO SECOND` без потери точности.

Воспользовавшись типом `DSINTERVAL_UNCONSTRAINED`, приведенный выше пример можно переписать следующим образом:

```
DECLARE
    mts INTERVAL DAY(9) TO SECOND(9);
    FUNCTION double_my_interval (
        mts_in IN DSINTERVAL_UNCONSTRAINED) RETURN DSINTERVAL_UNCONSTRAINED
    IS
    BEGIN
        RETURN mts_in * 2;
    END;
BEGIN
    mts := '100 0:0:0.123456789';
    DBMS_OUTPUT.PUT_LINE(mts);
    DBMS_OUTPUT.PUT_LINE(double_my_interval(mts));
END;
```

Результат будет таким:

```
+0000000100 00:00:00.123456789
+0000000200 00:00:00.246913578
```

Обратите внимание на то, что тип данных `DSINTERVAL_UNCONSTRAINED` используется дважды: один раз для задания типа формального параметра функции `double_my_interval`, а второй — для задания типа возвращаемого значения. В результате эту функцию можно вызывать для *любого* значения типа `INTERVAL DAY TO SECOND` без потери точности или ошибок.

Функции для работы с датой/временем

Oracle реализует набор функций для работы со значениями типа даты/времени. Многие из них уже встречались вам ранее в этой главе. Мы не будем подробно рассматривать все функции, но сводка в табл. 10.5 познакомит читателя с доступными возможностями. Если какие-то функции вас интересуют, обращайтесь за подробным описанием к справочнику *Oracle SQL Reference*.



Избегайте использования традиционных функций Oracle, обрабатывающих значения типа `DATE`, при работе с новыми типами данных `TIMESTAMP`. Вместо них следует по возможности использовать новые функции для типов `INTERVAL`. А `DATE`-функции должны использоваться только для обработки значений типа `DATE`.

Многие из приведенных в табл. 10.5 функций (в том числе `ADD_MONTHS`) получают значения типа `DATE`. При использовании таких функций с новыми типами данных `TIMESTAMP` могут возникнуть проблемы. Хотя любой из этих функций можно передать значение типа `TIMESTAMP`, Oracle неявно преобразует его к типу `DATE`, и только тогда функция выполнит свою задачу, например:

```
DECLARE
  ts TIMESTAMP WITH TIME ZONE;
BEGIN
  ts := SYSTIMESTAMP;

  --Обратите внимание: в значении переменной ts задаются
  --дробные секунды И часовой пояс.
  DBMS_OUTPUT.PUT_LINE(ts);

  --Изменение значения ts одной из встроенных функций.
  ts := LAST_DAY(ts);

  --Дробные секунды ПОТЕРЯНЫ, а часовой пояс заменен
  --часовым поясом сеанса.
  DBMS_OUTPUT.PUT_LINE(ts);
END;
```

Результат:

```
13-MAR-05 04.27.23.163826 PM -08:00
31-MAR-05 04.27.23.000000 PM -05:00
```

Таблица 10.5. Встроенные функции для работы со значениями даты/времени

| Имя | Описание |
|--------------------------------|--|
| <code>ADD_MONTHS</code> | Возвращает значение <code>DATE</code> , полученное в результате увеличения заданного значения <code>DATE</code> на заданное количество месяцев. См. «Сложение и вычитание интервалов» |
| <code>CAST</code> | Выполняет преобразования между типами данных — например, между <code>DATE</code> и различными значениями <code>TIMESTAMP</code> . См. « <code>CAST</code> и <code>EXTRACT</code> » |
| <code>CURRENT_DATE</code> | Возвращает текущую дату и время в часовом поясе сеанса как значение типа <code>DATE</code> |
| <code>CURRENT_TIMESTAMP</code> | Возвращает текущую дату и время в часовом поясе сеанса как значение типа <code>TIMESTAMP WITH TIME ZONE</code> |
| <code>DBTIMEZONE</code> | Возвращает смещение часового пояса базы данных относительно UTC в форме символической строки (например, <code>'-05:00'</code>). Часовой пояс базы данных используется только при работе со значениями типа <code>TIMESTAMP WITH LOCAL TIME ZONE</code> |
| <code>EXTRACT</code> | Возвращает значение <code>NUMBER</code> или <code>VARCHAR2</code> , содержащее конкретный элемент даты/времени — час, год или сокращение часового пояса. См. « <code>CAST</code> и <code>EXTRACT</code> » |
| <code>FROM_TZ</code> | Преобразует <code>TIMESTAMP</code> и данные часового пояса в значение типа <code>TIMESTAMP WITH TIME ZONE</code> |
| <code>LAST_DAY</code> | Возвращает последний день месяца для заданного входного значения <code>DATE</code> |
| <code>LOCALTIMESTAMP</code> | Возвращает текущую дату и время как значение типа <code>TIMESTAMP</code> в локальном часовом поясе |
| <code>MONTHS_BETWEEN</code> | Возвращает значение <code>NUMBER</code> , содержащее количество месяцев между двумя датами. См. «Вычисление интервала между двумя значениями <code>DATE</code> » |
| <code>NEW_TIME</code> | Преобразует значение типа <code>DATE</code> одного часового пояса в аналогичное значение другого пояса. Функция существует для сохранения совместимости со старым кодом; в новых приложениях следует использовать типы <code>TIMESTAMP WITH TIME ZONE</code> или <code>TIMESTAMP WITH LOCAL TIME ZONE</code> |
| <code>NEXT_DAY</code> | Возвращает дату первого дня недели, следующего за указанной датой |
| <code>NUMTODSINTERVAL</code> | Преобразует заданное количество дней, часов, минут или секунд (на ваш выбор) в значение типа <code>INTERVAL DAY TO SECOND</code> |

продолжение ⇨

Таблица 10.5 (продолжение)

| Имя | Описание |
|-----------------|---|
| NUMTOYMINTERVAL | Преобразует заданное количество годов и месяцев (на ваш выбор) в значение типа INTERVAL YEAR TO MONTH |
| ROUND | Возвращает значение типа DATE, округленное до заданных единиц |
| SESSIONTIMEZONE | Возвращает смещение часового пояса сеанса (относительно UTC) в форме символьной строки |
| SYS_EXTRACT_UTC | Преобразует значение типа TIMESTAMP WITH TIME ZONE в значение TIMESTAMP с той же датой и временем, нормализованное по времени UTC |
| SYSDATE | Возвращает текущую дату и время сервера Oracle как значение типа DATE |
| SYSTIMESTAMP | Возвращает текущую дату и время сервера Oracle как значение типа TIMESTAMP WITH TIME ZONE |
| TO_CHAR | Преобразует значение даты/времени в символьную строку. См. «Преобразование даты и времени» |
| TO_DATE | Преобразует символьную строку в значение типа DATE. См. «Преобразование даты и времени» |
| TO_DSINTERVAL | Преобразует символьную строку в значение типа INTERVAL DAY TO SECOND. См. «Преобразования интервалов» |
| TO_TIMESTAMP | Преобразует символьную строку в значение типа TIMESTAMP. См. «Преобразование даты и времени» |
| TO_TIMESTAMP_TZ | Преобразует символьную строку в значение типа TIMESTAMP WITH TIME ZONE. См. «Преобразование даты и времени» |
| TO_YMINTERVAL | Преобразует символьную строку в значение типа INTERVAL YEAR TO MONTH. См. «Преобразования интервалов» |
| TRUNC | Возвращает значение типа DATE, усеченное до заданных единиц |
| TZ_OFFSET | Возвращает смещение относительно UTC часового пояса, заданного названием или сокращением, в форме VARCHAR2 (например, '-05:00') |

В этом примере переменная `ts` содержит значение типа `TIMESTAMP WITH TIME ZONE`. Это значение неявно преобразуется в `DATE` при передаче `LAST_DAY`. Поскольку в типе `DATE` не сохраняются ни дробные части секунд, ни смещение часового пояса, эти части значения `ts` попросту отбрасываются. Результат `LAST_DAY` снова присваивается `ts`, что приводит к выполнению второго неявного преобразования — на этот раз `DATE` преобразуется в `TIMESTAMP WITH TIME ZONE`. Второе преобразование получает часовой пояс сеанса, поэтому в смещении часового пояса в итоговом значении мы видим `-05:00`.

Очень важно понимать эти преобразования... и избегать их. Несомненно, вы представляете, какие коварные ошибки могут появиться в программе из-за неосторожного использования функций `DATE` со значениями `TIMESTAMP`. Честно говоря, я не представляю, почему в Oracle встроенные функции `DATE` не были перегружены для нормальной работы с `TIMESTAMP`. Будьте осторожны!

11

Записи

Запись (record) представляет собой составную структуру данных; другими словами, запись состоит из нескольких полей, каждое из которых обладает собственным значением. Записи в программах PL/SQL очень похожи на строки в таблицах баз данных. Запись как целое не имеет собственного значения; однако значение имеет каждый ее компонент, или поле, а объединение их в единую запись позволяет хранить и обрабатывать все значения как одно целое. Записи сильно упрощают работу программиста, а переход от объявлений уровня полей к уровню записей повышает эффективность написания кода.

Записи в PL/SQL

Каждая строка таблицы состоит из одного или нескольких столбцов, которые могут содержать данные разных типов. Аналогичным образом запись состоит из одного или нескольких полей. Существует три способа определения записи, но после ее определения доступ к полям и их изменение всегда выполняются по одним и тем же правилам. Следующий блок демонстрирует объявление записи на базе таблицы базы данных. Допустим, имеется таблица для хранения информации о любимых книгах:

```
CREATE TABLE books (  
    book_id          INTEGER,  
    isbn             VARCHAR2(13),  
    title            VARCHAR2(200),  
    summary          VARCHAR2(2000),  
    author           VARCHAR2(200),  
    date_published   DATE,  
    page_count       NUMBER  
);
```

Далее мы можем легко создать запись на основании объявления этой таблицы, заполнить ее результатами запроса к базе данных, а затем работать со значениями отдельных столбцов через поля записи:

```
DECLARE  
    my_book books%ROWTYPE;  
BEGIN  
    SELECT *  
        INTO my_book  
        FROM books  
        WHERE title = 'Oracle PL/SQL Programming, 6th Edition';  
    IF my_book.author LIKE '%Feuerstein%'
```

продолжение ➤

```
THEN
    DBMS_OUTPUT.put_line ('Код ISBN: ' || my_book.isbn);
END IF;
END;
```

Записи также могут определяться на базе ранее определенного типа. Допустим, в базе данных нас интересует только имя автора и название книги. Вместо того чтобы использовать %ROWTYPE для объявления записи, мы создаем тип записи:

```
DECLARE
    TYPE author_title_rt IS RECORD (
        author books.author%TYPE
        ,title books.title%TYPE
    );
    l_book_info author_title_rt;
BEGIN
    SELECT author, title INTO l_book_info
    FROM books WHERE isbn = '978-1-449-32445-2';
```

Начнем с рассмотрения некоторых преимуществ использования записей. Далее будут более подробно представлены способы определения записей и примеры их использования в программах.

Преимущества использования записей

Структура данных записи представляет собой высокоуровневое средство адресации и обработки данных, определяемых в программах PL/SQL (в отличие от информации, хранящейся в таблицах баз данных). Представление данных в виде записи дает разработчикам определенные преимущества, о которых рассказывается в следующих разделах.

Абстракция данных

Используя метод абстракции при моделировании сущностей реального мира, вы намеренно обобщаете характеристики объекта, отделяете его от особенностей конкретной реализации, то есть стараетесь сформировать обобщенную модель объекта. При создании модулей конкретные операции модуля абстрагируются в его имени (и спецификации программы).

Чтобы создать запись, нужно выделить атрибуты или поля описываемого ею объекта, затем задать отношение между этими атрибутами и присвоить ему имя. Получившийся набор атрибутов, связанных определенным отношением, — это и есть запись.

Агрегатные операции

Организация информации в виде записей позволяет выполнять операции не только над отдельными атрибутами, но и над записью как единым целым. Такие агрегатные операции лишь усиливают абстракцию данных. На практике нередки случаи, когда интерес представляют не столько значения отдельных полей, сколько содержимое всей записи в целом.

Предположим, что по роду своей деятельности менеджер часто взаимодействует с различными компаниями. Информация о каждой из них хранится в базе данных в виде отдельной записи. Содержимое полей этой записи может быть разным, но для менеджера это не имеет особого значения. Например, его не интересует, сколько строк занимает адрес компании, две или три. Главное, чтобы информацию о компании можно было изменять, удалять или анализировать. Таким образом, организация данных в виде записи позволяет обрабатывать сведения о конкретной компании как единое целое, скрывая отдельные реквизиты, если они не нужны, но в то же время обеспечивает доступ к каждому из этих реквизитов.

Такой подход позволяет рассматривать данные как наборы объектов, к которым применимы определенные правила.

Компактность и простота кода

Использование записей помогает программисту писать более понятный и компактный код, который реже нуждается в модификации, содержит меньше комментариев и объявлений переменных; вместо множества разнородных переменных объявляется одна запись. Код получается более эстетичным, а его сопровождение требует меньших ресурсов.

Записи PL/SQL положительно влияют на качество кода как на этапе разработки, так и при дальнейшем сопровождении. Чтобы более полно использовать все возможности, предоставляемые этими замечательными структурами, старайтесь придерживаться следующих рекомендаций:

- **Создавайте записи, соответствующие курсорам.** Создавая в программе курсор, тут же добавьте соответствующую запись (исключение составляет лишь курсор цикла FOR). Данные из курсора всегда извлекайте только в запись, но не в отдельные переменные. В тех немногих случаях, когда для этого потребуются дополнительные усилия, вы не пожалеете о строгом соблюдении этого принципа и оцените элегантность полученного кода. А начиная с Oracle9i Release 2, записи можно использовать даже в DML-инструкциях!
- **Создавайте записи на основе таблиц.** Если в программе должны храниться данные, прочитанные из таблицы, создайте новую запись на базе таблицы (или воспользуйтесь заранее определенной записью). Такой подход позволяет объявить всего одну переменную вместо нескольких переменных. Что еще лучше, структура записи будет автоматически адаптироваться к изменениям в таблице при каждой компиляции.
- **Передавайте записи в качестве параметров.** Вызываемым процедурам по возможности передавайте не отдельные переменные, а целые записи. Тем самым вы снижаете вероятность изменения синтаксиса вызова процедур, благодаря чему программный код становится более стабильным.

Курсоры подробно описаны в главе 15. Однако курсоры так часто используются в сочетании с записями, что они будут задействованы во многих примерах этой главы.

Объявление записей

В PL/SQL существует три способа объявления записей:

- **Запись на основе таблицы.** Для объявления записи, каждое поле которой соответствует значению одноименного столбца таблицы, используется атрибут %ROWTYPE с именем таблицы. Пример объявления записи `one_book`, которая имеет ту же структуру, что и таблица `books`:

```
DECLARE
    one_book books%ROWTYPE;
```

- **Запись на основе курсора.** Для объявления записи на основе явно заданного курсора или курсора, представленного переменной, где каждое поле соответствует столбцу или именованному выражению в команде SELECT, предназначен атрибут %ROWTYPE. Вот пример объявления записи, которая имеет ту же структуру, что и явный курсор:

```
DECLARE
    CURSOR my_books_cur IS
    SELECT * FROM books
    WHERE author LIKE '%FEUERSTEIN%';

    one_SF_book my_books_cur%ROWTYPE;
```

- **Запись, определяемая программистом.** Чтобы создать запись с нуля, то есть явно определить каждое поле, задавая его имя и тип данных, следует воспользоваться командой `TYPE...RECORD`. Значением поля записи, определяемой программистом, может быть не только скалярное значение, но и другая запись. В следующем примере определяется тип записи, содержащей информацию о писательской карьере автора книги, и объявляется экземпляр записи этого типа:

```
DECLARE
    TYPE book_info_rt IS RECORD (
        author books.author%TYPE,
        category VARCHAR2(100),
        total_page_count POSITIVE);

    steven_as_author book_info_rt;
```

Обратите внимание: в объявлении записи пользовательского типа атрибут `%ROWTYPE` не указывается. Запись в данном случае имеет тип `book_info_rt`.

Общий формат объявления атрибута `%ROWTYPE` таков:

```
имя_записи [имя_схемы.]имя_объекта%ROWTYPE
[ DEFAULT := совместимая_запись ];
```

Указывать параметр `имя_схемы` не обязательно (если он не задан, то для разрешения ссылки используется схема, внутри которой компилируется данный код). В качестве параметра `имя_объекта` может использоваться явный курсор, курсорная переменная, таблица, представление или синоним. Также при объявлении можно указать необязательное значение по умолчанию, которое должно быть записью того же или совместимого типа.

Пример создания записи на основе курсорной переменной:

```
DECLARE
    TYPE book_rc IS REF CURSOR RETURN books%ROWTYPE;
    book_cv book_rc;

    one_book book_cv%ROWTYPE;
BEGIN
    ...
```

Запись также можно объявить и неявно — например, в цикле `FOR` с курсором. В следующем блоке кода раздел объявлений не содержит определения записи `book_rec`; PL/SQL автоматически объявляет запись с атрибутом `%ROWTYPE`:

```
BEGIN
    FOR book_rec IN (SELECT * FROM books)
    LOOP
        calculate_total_sales (book_rec);
    END LOOP;
END;
```

Поскольку вариант с `TYPE` является самым интересным и нетривиальным способом определения записи, рассмотрим его более подробно.

Записи, определяемые программистом

Записи, создаваемые на основе таблиц и курсоров, используются в тех случаях, когда в программе необходимо предусмотреть структуры для хранения определенных данных. Однако такие записи не охватывают всех потребностей программистов в составных типах данных. Допустим, программисту понадобится запись, структура которой не имеет ничего общего со структурой таблиц и представлений. Придется ли ему создавать фиктивный курсор только для того, чтобы получить запись нужной структуры? Конечно же, нет. В подобных ситуациях PL/SQL позволяет программисту самостоятельно определить структуру записи с помощью команды `TYPE..RECORD`.

Создавая запись самостоятельно, вы сами определяете количество ее полей, их имена и типы данных. Для объявления записи определяемого программистом типа нужно выполнить две операции.

1. Объявить или определить тип записи, задав ее структуру с помощью команды `TYPE`.
2. На основе этого типа объявляются реальные записи, которые затем можно будет использовать в программе.

Объявление типа записи

Тип записи определяется с помощью оператора `TYPE..RECORD`, в котором задается имя новой структуры и описания входящих в нее полей. Общий синтаксис этого оператора таков:

```
TYPE имя_типа IS RECORD
  (имя_поля1 тип_данных1 [[NOT NULL]:=|DEFAULT значение_по_умолчанию],
   имя_поля2 тип_данных2 [[NOT NULL]:=|DEFAULT значение_по_умолчанию],
   ...
   имя_поляN тип_данныхN [[NOT NULL]:=|DEFAULT значение_по_умолчанию]
  );
```

Здесь *имя_поляN* — имя N-го поля записи, а *тип_данныхN* — тип данных указанного поля. Поля записи могут относиться к любому из следующих типов:

- жестко запрограммированный скалярный тип данных (`VARCHAR2`, `NUMBER` и т. д.);
- подтип, определяемый программистом;
- тип, устанавливаемый на основе типа уже определенной структуры данных, например объявление с привязкой к атрибуту `%TYPE` или `%ROWTYPE` (в последнем случае мы создаем *вложенную запись*);
- тип коллекции PL/SQL — поле записи может быть списком и даже коллекцией;
- тип `REF CURSOR` (то есть поле содержит курсорную переменную).

Пример команды `TYPE...RECORD`:

```
TYPE company_rectype IS RECORD (
  comp# company.company_id%TYPE
  , list_of_names DBMS_SQL.VARCHAR2S
  , dataset SYS_REFCURSOR
);
```

Тип записи можно определить в разделе объявлений или в спецификации пакета. Последний подход позволяет ссылаться на тип записи в любом блоке PL/SQL, откомпилированном внутри схемы, к которой принадлежит пакет, а также в блоках PL/SQL, обладающих привилегиями `EXECUTE` для этого пакета.

Объявление записи

Создавая собственные пользовательские типы записей, вы можете использовать их в объявлении конкретных записей:

```
имя_записи тип_записи;
```

Здесь *имя_записи* — имя объявляемой записи, а *тип_записи* — имя типа, определенное в команде `TYPE...RECORD`. Например, чтобы создать запись с информацией о покупках клиента, сначала нужно определить ее тип:

```
PACKAGE customer_sales_pkg
IS
  TYPE customer_sales_rectype IS RECORD
    (customer_id  customer.customer_id%TYPE,
     customer_name customer.name%TYPE,
     total_sales  NUMBER (15,2)
    );
```

Запись представляет собой структуру из трех полей, которые содержат первичный ключ, имя клиента и общую сумму его покупок. Используя данный тип, можно объявлять записи на его основе:

```
DECLARE
    prev_customer_sales_rec customer_sales_pkg.customer_sales_rectype;
    top_customer_rec customer_sales_pkg.customer_sales_rectype;
```

Обратите внимание: чтобы показать, что это объявление записи, атрибут %ROWTYPE или другое ключевое слово не требуется. Атрибут %ROWTYPE нужен только для объявления записей, создаваемых на основе таблиц и курсоров.

Записи, созданные на основе этих типов, могут передаваться в аргументах процедур; для этого достаточно указать тип записи в качестве типа формального параметра:

```
PROCEDURE analyze_cust_sales (
    sales_rec_in IN customer_sales_pkg.customer_sales_rectype)
```

В определении типа записи для каждого поля наряду с его именем и типом можно задать значение по умолчанию. Для этой цели используется синтаксис DEFAULT или :=. Здесь имена полей внутри записи должны быть уникальными.

Примеры определений типов записей

Предположим, в программе объявлены подтип, курсор и ассоциативный массив¹:

```
SUBTYPE long_line_type IS VARCHAR2(2000);
```

```
CURSOR company_sales_cur IS
    SELECT name, SUM (order_amount) total_sales
    FROM company c, orders o
    WHERE c.company_id = o.company_id;
```

```
TYPE employee_ids_tabletype IS
    TABLE OF employees.employee_id%TYPE
    INDEX BY BINARY_INTEGER;
```

Затем в том же разделе объявлений определяются две записи.

- Задаваемая программистом запись представляет подмножество столбцов таблицы company и таблицу PL/SQL. Атрибут %TYPE используется для привязки полей записи к столбцам таблицы, третье поле записи является ассоциативным массивом с идентификационными кодами сотрудников):

```
TYPE company_rectype IS RECORD
    (company_id company.company_id%TYPE,
    company_name company.name%TYPE,
    new_hires_tab employee_ids_tabletype);
```

- Смешанная запись демонстрирует разные типы объявлений полей, в том числе ограничение NOT NULL, использование подтипа и атрибута %TYPE, значение по умолчанию, использование ассоциативного массива и вложенную запись:

```
TYPE mishmash_rectype IS RECORD
    (emp_number NUMBER(10) NOT NULL := 0,
    paragraph_text long_line_type,
    company_nm company.name%TYPE,
    total_sales company_sales.total_sales%TYPE := 0,
    new_hires_tab employee_ids_tabletype,
    prefers_nonsmoking_f1 BOOLEAN := FALSE,
    new_company_rec company_rectype
    );
```

¹ Как будет рассказано в главе 12, термином «ассоциативный массив» называется то, что ранее называлось «таблицей PL/SQL» или «индексной таблицей».

Как видите, PL/SQL обладает чрезвычайно гибкими средствами определения пользовательских структур данных. Записи могут включать таблицы, представления и команды SELECT. Как правило, они имеют произвольную структуру и содержат поля, которые представляют собой другие записи и ассоциативные массивы.

Обработка записей

Независимо от способа определения записи (на основе таблицы или курсора или с помощью явного определения TYPE...RECORD), приемы работы с ней всегда одинаковы. С записью можно работать либо как с «единым целым», либо с каждым из ее полей по отдельности.

Операции над записями

Обработка отдельной записи подразумевает, что в операциях отсутствуют ссылки на конкретные поля. В настоящее время PL/SQL поддерживает следующие операции над записями:

- копирование содержимого одной записи в другую (если они имеют совместимую структуру, то есть одинаковое количество полей одного или взаимопреобразуемых типов);
- присваивание записи значения NULL (простым оператором присваивания);
- передача записи в качестве аргумента;
- возврат записи функцией (команда RETURN).

Однако некоторые операции на уровне записей пока не поддерживаются.

- Если вы хотите проверить, содержат ли все поля записи значение NULL, использовать синтаксис IS NULL нельзя. Осуществить такую проверку можно лишь путем применения оператора IS NULL по отношению к каждому полю.
- Невозможно сравнить две записи в одной операции. Например, нельзя узнать, равны или нет две записи (то есть значения всех их полей), или же узнать, какая из записей больше. К сожалению, для того чтобы ответить на эти вопросы, нужно сравнить каждую пару полей. Далее в этой главе, в разделе «Сравнение записей» описывается утилита, автоматически генерирующая код для сравнения двух записей.
- Только в Oracle9i Release 2 появилась возможность добавления новых записей в таблицу базы данных. В предыдущих версиях системы значение каждого поля приходилось записывать в соответствующий столбец таблицы отдельно. Более полная информация об инструкциях языка DML, предназначенных для обработки записей, содержится в главе 14.

Операции уровня записей могут выполняться над любыми записями с совместимыми структурами. Иначе говоря, у обрабатываемых записей должно быть одинаковое количество полей, причем эти поля должны иметь одинаковые или взаимопреобразуемые типы данных. Предположим, имеется такая таблица:

```
CREATE TABLE cust_sales_roundup (  
  customer_id NUMBER (5),  
  customer_name VARCHAR2 (100),  
  total_sales NUMBER (15,2)  
)
```

Три объявленные ниже записи имеют совместимую структуру и их можно «смешивать» в разных операциях:

```
DECLARE  
  cust_sales_roundup_rec cust_sales_roundup%ROWTYPE;  
  
  CURSOR cust_sales_cur IS SELECT * FROM cust_sales_roundup;  
  cust_sales_rec cust_sales_cur%ROWTYPE;
```

```
TYPE customer_sales_rectype IS RECORD
  (customer_id NUMBER(5),
   customer_name customer.name%TYPE,
   total_sales NUMBER(15,2)
  );
```

```
preferred_cust_rec customer_sales_rectype;
```

```
BEGIN
```

```
-- Присвоить содержимое одной записи другой.
```

```
cust_sales_roundup_rec := cust_sales_rec;
```

```
preferred_cust_rec := cust_sales_rec;
```

```
END;
```

Рассмотрим еще несколько примеров выполнения операций на уровне записей.

- Запись можно инициализировать при объявлении, присвоив содержимое другой, совместимой с ней записи. В следующем фрагменте локальной переменной присваивается запись, переданная процедуре в аргументе IN. Теперь значения отдельных полей записи можно изменять:

```
PROCEDURE compare_companies
  (prev_company_rec IN company%ROWTYPE)
IS
  curr_company_rec company%ROWTYPE := prev_company_rec;
BEGIN
  ...
END;
```

- В следующем примере сначала создается новый тип записи и объявляется запись этого типа. Затем создается второй тип записи, для которого в качестве типа единственного столбца устанавливается первый тип. В определении данного типа столбец-запись инициализируется переменной-записью:

```
DECLARE
  TYPE first_rectype IS RECORD (var1 VARCHAR2(100) := 'WHY NOT');
  first_rec first_rectype;
  TYPE second_rectype IS RECORD (nested_rec first_rectype := first_rec);
BEGIN
  ...
END;
```

Разумеется, операцию присваивания можно осуществить и в разделе выполнения. Ниже объявляются две разные записи типа `rain_forest_history`, и информация из первой из них присваивается второй записи:

```
DECLARE
  prev_rain_forest_rec rain_forest_history%ROWTYPE;
  curr_rain_forest_rec rain_forest_history%ROWTYPE;
BEGIN
  ... Инициализация записи prev_rain_forest_rec ...

  -- Копирование данных первой записи во вторую
  curr_rain_forest_rec := prev_rain_forest_rec;
```

- В результате выполнения такой операции значение каждого поля первой записи присваивается соответствующему полю второй записи. Значения полям можно было бы присваивать и по отдельности, но согласитесь: так гораздо удобнее. Поэтому старайтесь обрабатывать записи как одно целое — это позволит сэкономить время, упростить программный код и облегчить его сопровождение.
- Для перемещения данных строки таблицы в запись достаточно одной операции выборки. Примеры:

```
DECLARE
  /*
  || Объявляем курсор, а затем при помощи атрибута %ROWTYPE
  || определяем запись на его основе
```

```

*/
CURSOR cust_sales_cur IS
    SELECT customer_id, customer_name, SUM (total_sales) tot_sales
    FROM cust_sales_roundup
    WHERE sold_on < ADD_MONTHS (SYSDATE, -3)
    GROUP BY customer_id, customer_name;
cust_sales_rec cust_sales_cur%ROWTYPE;
BEGIN
    /* Перемещаем значения из курсора непосредственно в запись */

    OPEN cust_sales_cur;
    FETCH cust_sales_cur INTO cust_sales_rec;
    CLOSE cust_sales_cur;

```

- В следующем блоке сначала задается тип записи, который соответствует данным, возвращаемым неявным курсором, а затем производится извлечение данных непосредственно в запись:

```

DECLARE
    TYPE customer_sales_rectype IS RECORD
        (customer_id customer.customer_id%TYPE,
         customer_name customer.name%TYPE,
         total_sales NUMBER (15,2)
        );
    top_customer_rec customer_sales_rectype;
BEGIN
    /* Загрузка значений непосредственно в запись: */
    SELECT customer_id, customer_name, SUM (total_sales)
    INTO top_customer_rec
    FROM cust_sales_roundup
    WHERE sold_on < ADD_MONTHS (SYSDATE, -3)
    GROUP BY customer_id, customer_name;

```

- В следующем примере одна операция присваивает всем полям записи значение NULL:

```

/* Файл в Сети: record_assign_null.sql */
FUNCTION dept_for_name (
    department_name_in IN departments.department_name%TYPE
)
    RETURN departments%ROWTYPE
IS
    l_return departments%ROWTYPE;

    FUNCTION is_secret_department (
        department_name_in IN departments.department_name%TYPE
    )
        RETURN BOOLEAN
    IS
    BEGIN
        RETURN CASE department_name_in
            WHEN 'VICE PRESIDENT' THEN TRUE
            ELSE FALSE
        END;
    END is_secret_department;
BEGIN
    SELECT *
    INTO l_return
    FROM departments
    WHERE department_name = department_name_in;

    IF is_secret_department (department_name_in)
    THEN
        l_return := NULL;
    END IF;

    RETURN l_return;
END dept_for_name;

```

По возможности старайтесь работать с записями на агрегатном уровне, то есть обрабатывать каждую из них как единое целое, не разбивая на отдельные поля. Благодаря этому результирующий код получится более простым, понятным и стабильным. Конечно, существует множество ситуаций, когда требуется работать лишь с отдельными полями записи. Давайте посмотрим, как это делается.

Операции над отдельными полями

Чтобы получить доступ к отдельному полю записи (для выполнения операции чтения или изменения), используйте точечный синтаксис, как при идентификации столбца таблицы базы данных:

```
[имя_схемы.]имя_пакета.имя_записи.имя_поля
```

Имя пакета необходимо указывать только в том случае, если запись определена в спецификации другого пакета (не того, с которым вы работаете в настоящее время). А имя схемы потребуется определить лишь при условии, что пакет принадлежит не той схеме, где компилируется данный код.

После того как поле записи будет идентифицировано посредством точечного синтаксиса, хранящееся в нем значение можно обрабатывать точно так же, как любую другую переменную. Рассмотрим несколько примеров.

Оператор присваивания `:=` изменяет значение в конкретном поле записи. В первом примере обнуляется поле `total_sales`. Во втором примере вызывается функция, которая возвращает значение флага `output_generated` (ему присваивается `TRUE`, `FALSE` или `NULL`):

```
BEGIN
  top_customer_rec.total_sales := 0;
  report_rec.output_generated := check_report_status (report_rec.report_id);
END;
```

Код следующего примера создает запись на основе таблицы `rain_forest_history`, заполняет ее поля значениями и возвращает запись в ту же таблицу:

```
DECLARE
  rain_forest_rec rain_forest_history%ROWTYPE;
BEGIN
  /* Установка значений полей записи */
  rain_forest_rec.country_code := 1005;
  rain_forest_rec.analysis_date := ADD_MONTHS (TRUNC (SYSDATE), -3);
  rain_forest_rec.size_in_acres := 32;
  rain_forest_rec.species_lost := 425;

  /* Вставка в таблицу строки значений из записи */
  INSERT INTO rain_forest_history
    (country_code, analysis_date, size_in_acres, species_lost)
  VALUES
    (rain_forest_rec.country_code,
     rain_forest_rec.analysis_date,
     rain_forest_rec.size_in_acres,
     rain_forest_rec.species_lost);
  ...
END;
```

Обратите внимание, что полю `analysis_date` можно присвоить любое допустимое выражение типа `DATE`. Сказанное относится и к другим полям, а также более сложным структурам.

Начиная с Oracle9i Release 2, появилась возможность выполнения вставки уровня записей, в результате чего приведенная инструкция `INSERT` упрощается до следующего вида:

```
INSERT INTO rain_forest_history
  (country_code, analysis_date, size_in_acres, species_lost)
VALUES rain_forest_rec;
```

DML-инструкции уровня записей (как для вставки, так и для обновления) подробно описаны в главе 14.

Операции уровня полей с вложенными записями

Предположим, у нас имеется вложенная структура записей, то есть одно из полей «внешней» записи в действительности является другой записью. В следующем примере сначала определяется тип записи для хранения всех элементов телефонного номера (`phone_rectype`), а затем — тип записи, в которой объединяются в единую структуру `contact_set_rectype` несколько телефонных номеров одного человека:

```
DECLARE
    TYPE phone_rectype IS RECORD
        (intl_prefix  VARCHAR2(2),
         area_code    VARCHAR2(3),
         exchange     VARCHAR2(3),
         phn_number   VARCHAR2(4),
         extension    VARCHAR2(4)
        );

    -- Каждое поле представляет собой вложенную запись
    TYPE contact_set_rectype IS RECORD
        (day_phone#   phone_rectype,
         eve_phone#   phone_rectype,
         fax_phone#   phone_rectype,
         home_phone#  phone_rectype,
         cell_phone#  phone_rectype
        );

    auth_rep_info_rec contact_set_rectype;
BEGIN
```

Для ссылки на поля вложенной записи используется точечный синтаксис, причем таким образом можно обращаться к полям любого уровня вложенности. Имена вложенных структур перечисляются последовательно и разделяются точками. В приведенном ниже фрагменте кода в поле междугородного кода факса заносится код из поля домашнего телефона:

```
auth_rep_info_rec.fax_phone#.area_code :=
    auth_rep_info_rec.home_phone#.area_code;
```

Операции уровня полей с записями на базе пакетов

Последний пример демонстрирует принцип использования ссылок на записи и типы записей, объявленные в пакетах. Предположим, пользователь формирует список книг, которые он планирует прочесть за время летнего отпуска. Для этого он создает следующую спецификацию пакета:

```
CREATE OR REPLACE PACKAGE summer
IS
    TYPE reading_list_rt IS RECORD (
        favorite_author VARCHAR2 (100),
        title            VARCHAR2 (100),
        finish_by        DATE);

    must_read reading_list_rt;
    wives_favorite reading_list_rt;
END summer;

CREATE OR REPLACE PACKAGE BODY summer
IS
BEGIN -- Раздел инициализации пакета
    must_read.favorite_author := 'Tepper, Sheri S.';
    must_read.title := 'Gate to Women's Country';
END summer;
```

После того как этот пакет будет откомпилирован в базе данных, список литературы можно будет построить следующим образом:

```
DECLARE
    first_book summer.reading_list_rt;
    second_book summer.reading_list_rt;
BEGIN
    summer.must_read.finish_by := TO_DATE ('01-AUG-2009', 'DD-MON-YYYY');
    first_book := summer.must_read;

    second_book.favorite_author := 'Hobb, Robin';
    second_book.title := 'Assassin's Apprentice';
    second_book.finish_by := TO_DATE ('01-SEP-2009', 'DD-MON-YYYY');
END;
```

Мы объявляем две записи для представления информации о книге. Сначала устанавливается значение поля `finish_by` объявленной в пакете `summer` записи `must_read` (обратите внимание на синтаксис *пакет.запись.поле*), затем запись присваивается переменной, представляющей первую книгу из числа запланированных для чтения. После этого значения присваиваются отдельным полям записи, относящейся ко второй книге.

При работе со встроенным пакетом `UTL_FILE`, предназначенным для выполнения файлового ввода/вывода в PL/SQL, необходимо следовать тем же правилам обработки записей. Объявление типа данных `UTL_FILE.FILE_TYPE` фактически является определением типа записи. Таким образом, при объявлении дескриптора файла вы на самом деле объявляете запись типа, определяемого в пакете:

```
DECLARE
    my_file_id UTL_FILE.FILE_TYPE;
```

Сравнение записей

Как узнать, равны ли две записи, то есть совпадают ли их соответствующие поля? Было бы неплохо, если бы PL/SQL позволял выполнять непосредственное сравнение. Например:

```
DECLARE
    first_book summer.reading_list_rt := summer.must_read;
    second_book summer.reading_list_rt := summer.wifes_favorite;
BEGIN
    IF first_book = second_book /* НЕ ПОДДЕРЖИВАЕТСЯ! */
    THEN
        lots_to_talk_about;
    END IF;
END;
```

К сожалению, это невозможно. Чтобы сравнить две записи, вам придется написать код для сравнения всех их полей по отдельности. Если количество полей невелико, сделать это довольно просто. Например, в случае записей из предыдущего примера сравнение выполняется следующим образом:

```
DECLARE
    first_book summer.reading_list_rt := summer.must_read;
    second_book summer.reading_list_rt := summer.wifes_favorite;
BEGIN
    IF first_book.favorite_author = second_book.favorite_author
    AND first_book.title = second_book.title
    AND first_book.finish_by = second_book.finish_by
    THEN
        lots_to_talk_about;
    END IF;
END;
```


Однако здесь возникает одна проблема. Если вы хотите указать, что две записи со значениями NULL должны считаться равными, оператор сравнения необходимо изменить следующим образом:

```
(first_book.favorite_author = second_book.favorite_author
OR ( first_book.favorite_author IS NULL AND
    second_book.favorite_author IS NULL )
```

Как видите, код получается довольно громоздким. Правда, хорошо было бы генерировать его автоматически? Оказывается, это и в самом деле возможно — по крайней мере для тех записей, которые определены с атрибутом %ROWTYPE на основе таблиц или представлений. В таком случае имена полей можно получить из представления ALL_TAB_COLUMNS в словаре данных, а затем вывести сгенерированный код на экран или в файл.

К счастью, вам не придется самостоятельно разбираться во всех тонкостях. Готовый генератор кода, разработанный Дэном Спенсером (Dan Spencer), можно найти на сайте книги в файле gen_record_comparison.pkg.

Триггерные псевдозаписи

При программировании триггеров для конкретной таблицы Oracle предоставляет в ваше распоряжение две структуры, OLD и NEW, которые являются *псевдозаписями*. По формату эти структуры сходны с записями на базе таблиц, объявленными с атрибутом %ROWTYPE: они также содержат поле для каждого столбца таблицы:

- OLD — псевдозапись содержит значения всех столбцов таблицы до начала текущей транзакции.
- NEW — псевдозапись содержит новые значения всех столбцов, которые будут помещены в таблицу при завершении текущей транзакции.

При использовании ссылок на OLD и NEW в теле триггера перед идентификаторами ставится двоеточие, а в условии WHEN оно не используется. Пример:

```
TRIGGER check_raise
AFTER UPDATE OF salary
ON employee
FOR EACH ROW
WHEN (OLD.salary != NEW.salary) OR
      (OLD.salary IS NULL AND NEW.salary IS NOT NULL) OR
      (OLD.salary IS NOT NULL AND NEW.salary IS NULL)
BEGIN
  IF :NEW.salary > 100000 THEN ...
```

В главе 19 более подробно описаны возможности использования псевдозаписей OLD и NEW в триггерах баз данных. В частности, в этой главе представлены многие ограничения, связанные с использованием OLD и NEW.

%ROWTYPE и невидимые столбцы (Oracle Database 12c)

В версии 12.1 появилась возможность определения *невидимых* (INVISIBLE) столбцов в реляционных таблицах. Если вы захотите вывести или присвоить значение невидимого столбца, его имя необходимо задать явно. Пример определения невидимого столбца в таблице:

```
CREATE TABLE my_table (i INTEGER, d DATE, t TIMESTAMP INVISIBLE)
```

Чтобы сделать невидимый столбец видимым, выполните соответствующую команду ALTER TABLE:

```
ALTER TABLE my_table MODIFY t VISIBLE
```

Синтаксис `SELECT *` не отображает столбцы `INVISIBLE`. Но если включить столбец `INVISIBLE` в список выборки команды `SELECT`, столбец будет выводиться. Значение столбца `INVISIBLE` нельзя явно задать в условии `VALUES` команды `INSERT`; такие столбцы должны явно задаваться в атрибутах `%ROWTYPE`.

Например, при попытке откомпилировать следующий блок выдается ошибка *PLS-00302: component 'T' must be declared*:

```
DECLARE
    /* Запись с двумя полями: i и d */
    l_data my_table%ROWTYPE;
BEGIN
    SELECT * INTO l_data FROM my_table;
    DBMS_OUTPUT.PUT_LINE ('t = ' || l_data.t);
END;
/
```

Так как столбец `T` является невидимым, запись, объявленная с `%ROWTYPE`, содержит только два поля с именами `I` и `D`. Мы не можем обратиться к полю с именем `T`. Но если сделать столбец видимым, Oracle создаст для него поле в записи, объявленной с использованием `%ROWTYPE`. Это также означает, что после того, как невидимый столбец станет видимым, Oracle изменит статус всех программных модулей, объявляющих записи с использованием синтаксиса `%ROWTYPE` для этой таблицы, на недействительный (`INVALID`).

12 Коллекции

Коллекцией называется структура данных, по своей функциональности сходная со списком или одномерным массивом. В сущности, коллекция — ближайший аналог традиционных массивов в программах PL/SQL. Эта глава поможет вам решить, какой из трех разных типов коллекций (ассоциативный массив, вложенная таблица, `VARRAY`) лучше всего соответствует потребностям вашей программы. Также вы познакомитесь с примерами определения этих структур и работы с ними.

Несколько типичных ситуаций для применения коллекций:

- **Ведение списков данных в программах.** Вероятно, это самое частое применение коллекций в программах. Да, с таким же успехом можно воспользоваться реляционными таблицами, глобальными временными таблицами (работа с которыми требует частых переключений контекста) или строками с разделителями, но коллекции чрезвычайно эффективны, а код работы с ними получается очень выразительным и простым в сопровождении.
- **Ускорение многострочных операций SQL на порядок и более.** Использование коллекций в сочетании с конструкциями `FORALL` и `BULK COLLECT` радикально повышает производительность многострочных операций SQL. Эти операции подробно описаны в главе 21.
- **Кэширование информации базы данных.** Коллекции хорошо подходят для кэширования статической информации, которая часто запрашивается в ходе одного сеанса (или просто многократно запрашивается в ходе выполнения одной программы) для повышения производительности поиска.

Как ни странно, лишь относительно немногие разработчики знают о существовании коллекций и пользуются ими в своей работе. Прежде всего, это объясняется относительной сложностью коллекций. Три разных типа коллекций, многоэтапная процедура их использования и определения, использование в программах PL/SQL и базах данных объектов, более сложный синтаксис по сравнению с обычными переменными: все эти факторы снижают популярность коллекций.

В этой главе я постарался по возможности полно изложить тему коллекций, обойтись без избыточности в описании разных типов коллекций и дать полезные рекомендации по их использованию. В результате глава получилась длинной, но я уверен, что вы найдете в ней много полезного. Ниже приведена краткая сводка содержимого.

- **Общие сведения о коллекциях** — мы начнем со знакомства с основными понятиями: описанием разных видов коллекций, терминологии коллекций, содержательными примерами для каждой разновидности коллекций, рекомендациями по выбору

коллекции для конкретных ситуаций. Даже если вы ограничитесь чтением только этого раздела, скорее всего, вы сможете написать базовую логику коллекций. Однако я настоятельно рекомендую читать дальше!

- **Методы коллекций** — затем мы изучим многочисленные методы (процедуры и функции), которые Oracle предоставляет для просмотра и выполнения операций с содержимым коллекций. Практически любое применение коллекций сопряжено с использованием этих методов, поэтому вы должны хорошо понимать, что и как они делают.
- **Работа с коллекциями** — от «азов» можно перейти к нетривиальным аспектам работы с коллекциями, включая процесс инициализации, необходимый для вложенных таблиц и VARRAY, разные способы заполнения коллекций и обращения к их данным, операции со столбцами на языке SQL и коллекции, индексируемые строками.
- **Операции мультимножеств со вложенными таблицами** — в Oracle Database 10g реализация вложенных таблиц была дополнена возможностью выполнения операций множеств с содержимым вложенных таблиц (объединение, пересечение, вычитание и т. д.). Две вложенные таблицы можно сравнить на равенство и неравенство.
- **Сопровождение коллекций на уровне схемы** — вложенные таблицы и типы VARRAY можно определять в самой базе данных. В базе данных содержатся представления словарей данных, используемые для сопровождения этих типов.

Знакомство с коллекциями

Начнем с обзора основных концепций и терминологии коллекций, описания разных типов коллекций и примеров.

Концепции и терминология

Следующие пояснения помогут читателю быстрее освоить эти структуры данных.

- **Элементы и индексы.** Коллекция состоит из множества элементов (фрагментов данных), причем каждый элемент находится в определенной позиции списка, то есть обладает определенным индексом. Иногда элементы называются «*строками*», а индексы — «*номераами строк*».
- **Тип коллекции.** Каждая переменная, представляющая коллекцию в программе, должна быть объявлена на основании заранее определенного *типа* коллекции. Как упоминалось ранее, коллекции делятся на три широкие категории: ассоциативные массивы, вложенные таблицы и VARRAY. В этих категориях существуют конкретные типы, которые определяются командой TYPE в разделе объявлений блока. Далее программист объявляет и использует экземпляры этих типов в своих программах.
- **Коллекция, или экземпляр коллекции.** Этот термин может иметь несколько значений:
 - переменная PL/SQL типа ассоциативного массива, вложенной таблицы или VARRAY;
 - столбец таблицы, где хранятся значения типа вложенной таблицы или массива VARRAY.

Независимо от конкретного использования коллекция всегда остается списком элементов.

Экземпляром коллекции называется экземпляр конкретного типа коллекции.

Отчасти из-за синтаксиса и названий, которые были выбраны для поддержки коллекций Oracle, коллекции также иногда называются *массивами* и *таблицами*.

- **Однородные элементы.** Все элементы коллекции должны относиться к одному типу, то есть коллекция является *однородной*. Впрочем, этот тип данных может быть составным; например, можно объявить коллекцию, состоящую из записей, то есть таблицу. Начиная с Oracle9i, поддерживается возможность объявления многоуровневых коллекций: полем коллекции служит коллекция или запись, полем которой, в свою очередь, также является коллекция или запись и т. д.
- **Одномерная коллекция.** В каждой строке коллекции имеется всего одно поле, и с этой точки зрения она похожа на одномерный массив. Нельзя объявить коллекцию, на элементы которой можно было бы ссылаться следующим образом:

```
my_collection (10, 44)
```

Двумерные структуры в настоящее время напрямую не поддерживаются, однако вы можете создать многомерный массив, объявляя коллекцию коллекцией, — синтаксис будет примерно таким:

```
my_collection (44) (10)
```

- **Ограниченная и неограниченная коллекция.** Коллекция называется ограниченной, если заранее определены границы возможных значений индексов (номеров) ее элементов. Если же верхняя или нижняя граница номеров элементов не указана, то коллекция называется неограниченной. Коллекции типа `VARRAY` (массивы переменной длины) всегда ограничены. При определении такой коллекции следует указать максимальное количество ее элементов (номер первого элемента всегда равен 1). Вложенные таблицы и ассоциативные массивы ограничиваются только теоретически. В тексте они будут описываться как неограниченные, потому что с теоретической точки зрения максимальное количество элементов в них может быть произвольным.
- **Разреженные и плотные коллекции.** Коллекция (или массив, или список) называется *плотной*, если все ее элементы, от первого до последнего, определены и каждому из них присвоено некоторое значение (таковым может быть и `NULL`). Коллекция считается *разреженной*, если отдельные ее элементы отсутствуют, как в рассмотренном выше примере. Не обязательно определять все элементы коллекции и заполнять ее полностью. Массивы `VARRAY` всегда являются плотными. Вложенные таблицы первоначально всегда плотные, но по мере удаления некоторых элементов становятся разреженными. Ассоциативные массивы могут быть как разреженными, так и плотными в зависимости от способа их заполнения.
Разреженность — это очень ценное свойство, позволяющее добавлять элементы в коллекцию по первичному ключу или другим ключевым данным (например, номеру записи). Таким образом можно задать определенный порядок следования данных в коллекции или значительно ускорить их поиск.
- **Целочисленное индексирование.** К любому элементу коллекции можно обратиться по номеру, который представляет собой целочисленное значение. Объявление ассоциативного массива явно выражает это требование условием `INDEX BY`, но правило действует и для других типов коллекций.
- **Строковое индексирование.** Начиная с Oracle9i Release 2, в качестве индексов ассоциативных массивов можно использовать не только номера, но и символьные строки (в настоящее время до 32 Кбайт длиной). Эта возможность не поддерживается ни для вложенных таблиц, ни для массивов `VARRAY`.
- **Внешняя таблица.** Так называют таблицу, содержащую столбец типа вложенной таблицы или массива `VARRAY`.
- **Внутренняя таблица.** Так принято называть коллекцию, содержащуюся в столбце таблицы.

- **Вспомогательная таблица.** Физическая таблица, создаваемая Oracle для хранения внутренней таблицы (столбца, содержащего вложенную таблицу).

Разновидности коллекций

Как упоминалось ранее, Oracle поддерживает три разновидности коллекций. Несмотря на ряд общих характеристик, все эти разновидности обладают специфическими особенностями, которые будут охарактеризованы ниже.

- **Ассоциативные массивы.** Это одномерные неограниченные разреженные коллекции, состоящие из однородных элементов, доступные только в PL/SQL. Ранее в PL/SQL 2 (из поставки Oracle7) они назывались *таблицами PL/SQL*, а в Oracle8 и Oracle8i — *индексируемыми таблицами* (поскольку при объявлении такой коллекции приходилось явно указывать, что она «индексируется» номером строки). В Oracle9i Release 1 они были названы *ассоциативными массивами*. Этот термин выбран потому, что предложение **INDEX BY** может использоваться для индексирования содержимого коллекций посредством значений типа **VARCHAR2** или **PLS_INTEGER**.
- **Вложенные таблицы.** Так называются одномерные несвязанные коллекции, также состоящие из однородных элементов. Первоначально они заполняются полностью, но позднее из-за удаления некоторых элементов могут стать разреженными. Вложенные таблицы могут определяться и в PL/SQL, и в базах данных (например, в качестве столбцов таблиц). Вложенные таблицы представляют собой *мультимножества*, то есть элементы вложенной таблицы не упорядочены.
- **Массив типа VARRAY.** Подобно двум другим типам коллекций, массивы **VARRAY** (массивы переменной длины) также являются одномерными коллекциями, состоящими из однородных элементов. Однако их размер всегда ограничен, и они не бывают разреженными. Как и вложенные таблицы, массивы **VARRAY** используются и в PL/SQL, и в базах данных. Однако порядок их элементов при сохранении и выборке, в отличие от вложенных таблиц, сохраняется.

Примеры коллекций

В этом разделе представлены относительно простые примеры всех разновидностей коллекций с описанием их основных характеристик.

Ассоциативный массив

В следующем примере мы объявляем тип ассоциативного массива, а затем коллекцию, основанную на этом типе. Коллекция заполняется четырьмя элементами, после чего мы перебираем ее содержимое и выводим символьные строки. Более подробное объяснение приведено после листинга.

```

1  DECLARE
2      TYPE list_of_names_t IS TABLE OF person.first_name%TYPE
3          INDEX BY PLS_INTEGER;
4      happyfamily list_of_names_t;
5      l_row PLS_INTEGER;
6  BEGIN
7      happyfamily (2020202020) := 'Eli';
8      happyfamily (-15070) := 'Steven';
9      happyfamily (-90900) := 'Chris';
10     happyfamily (88) := 'Veve';
11
12     l_row := happyfamily.FIRST;
13
14     WHILE (l_row IS NOT NULL)
15     LOOP
```

```

16      DBMS_OUTPUT.put_line (happyfamily (1_row));
17      1_row := happyfamily.NEXT (1_row);
18  END LOOP;
19  END;

```

Результат:

```

Chris
Steven
Veva
Eli

```

| Строки | Описание |
|--------|--|
| 2–3 | Объявление ассоциативного массива TYPE с характерной секцией INDEX BY. Коллекция, созданная на основе этого типа, содержит список строк, каждая из которых может достигать по длине столбца first_name таблицы person |
| 4 | Объявление коллекции happyfamily на базе типа list_of_names_t |
| 9–10 | Заполнение коллекции четырьмя именами. Обратите внимание: мы можем использовать любые целочисленные значения по своему усмотрению. Номера строк в ассоциативном массиве не обязаны быть последовательными; они даже могут быть отрицательными! Никогда не пишите код с произвольно выбранными, непонятными значениями индексов! Этот пример всего лишь демонстрирует гибкость ассоциативных массивов |
| 12 | Вызов метода FIRST (функция, «прикрепленная» к коллекции) для получения первого (минимального) номера строки в коллекции |
| 14–18 | Перебор содержимого коллекции в цикле WHILE, с выводом каждой строки. В строке 17 вызывается метод NEXT, который переходит от текущего элемента к следующему без учета промежуточных пропусков |

Вложенная таблица

Следующий пример начинается с объявления типа вложенной таблицы как типа уровня схемы. На основании этого типа в блоке PL/SQL объявляются три вложенные таблицы. Имена членов семьи сохраняются во вложенной таблице happyfamily, имена детей — во вложенной таблице children, после чего команда MULTISET EXCEPT (появившаяся в Oracle10g) используется для извлечения из вложенной таблицы happyfamily только имен родителей. Более подробное объяснение приводится после листинга.

REM Раздел A

```

SQL> CREATE TYPE list_of_names_t IS TABLE OF VARCHAR2 (100);
2 /
Type created.

```

REM Раздел B

```

1  DECLARE
2      happyfamily    list_of_names_t := list_of_names_t ();
3      children       list_of_names_t := list_of_names_t ();
4      parents        list_of_names_t := list_of_names_t ();
5  BEGIN
6      happyfamily.EXTEND (4);
7      happyfamily (1) := 'Eli';
8      happyfamily (2) := 'Steven';
9      happyfamily (3) := 'Chris';
10     happyfamily (4) := 'Veva';
11
12     children.EXTEND;
13     children (1) := 'Chris';
14     children.EXTEND;
15     children (2) := 'Eli';
16
17     parents := happyfamily MULTISET EXCEPT children;
18
19     FOR 1_row IN parents.FIRST .. parents.LAST
20     LOOP

```

```

21         DBMS_OUTPUT.put_line (parents (1_row));
22     END LOOP;
23 END;

```

Результат:

```

Steven
Veva

```

| Строки | Описание |
|----------|---|
| Раздел A | Команда CREATE TYPE создает тип вложенной таблицы в самой базе данных. Такое решение позволяет объявлять вложенные таблицы в любой схеме, обладающей разрешениями EXECUTE для этого типа. Также можно объявлять столбцы в реляционных таблицах этого типа |
| 2–4 | Объявление трех разных вложенных таблиц, созданных на основе типа уровня схемы. Обратите внимание: в каждом случае для инициализации вложенной таблицы вызывается функция-конструктор. Имя этой функции всегда совпадает с именем типа, а ее код автоматически генерируется Oracle. Вложенную таблицу необходимо инициализировать перед использованием |
| 6 | Вызов метода EXTEND «выделяет место» во вложенной таблице для данных членов семьи. В отличие от ассоциативных массивов, во вложенных таблицах необходимо явно выполнить операцию создания элемента, прежде чем размещать в нем данные |
| 7–10 | Заполнение коллекции happyfamily именами членов семьи |
| 12–15 | Заполнение коллекции children. В данном случае данные добавляются в коллекцию по строкам |
| 17 | Чтобы получить информацию о родителях, достаточно убрать из happyfamily данные children. Эта задача особенно легко решается, начиная с версии Oracle10g, с появлением высокоуровневых команд для работы с множествами вроде MULTISET EXCEPT (близкий аналог коллекции SQL MINUS). Обратите внимание: перед заполнением parents вызывать метод EXTEND не нужно. База данных делает это автоматически |
| 19–22 | Поскольку операция MULTISET EXCEPT плотно заполняет коллекцию parents, для перебора содержимого коллекции можно воспользоваться циклом FOR со счетчиком. При использовании этой конструкции с разреженной коллекцией произойдет исключение NO_DATA_FOUND |

VARRAY

Следующий пример демонстрирует использование типов VARRAY в качестве столбцов реляционной таблицы. Сначала мы объявляем два разных типа VARRAY уровня схемы, после чего создаем реляционную таблицу family с двумя столбцами VARRAY. Наконец, в коде PL/SQL заполняются две локальные коллекции, используемые при выполнении операции INSERT с таблицей family. Более подробное объяснение приводится после листинга.

```

REM Раздел A
SQL> CREATE TYPE first_names_t IS VARRAY (2) OF VARCHAR2 (100);
2 /
Type created.

```

```

SQL> CREATE TYPE child_names_t IS VARRAY (1) OF VARCHAR2 (100);
2 /
Type created.

```

```

REM Раздел B
SQL> CREATE TABLE family (
2     surname VARCHAR2(1000)
3     , parent_names first_names_t
4     , children_names child_names_t
5 );

```

Table created.

```

REM Раздел C
SQL>

```



```

1 DECLARE
2     parents    first_names_t := first_names_t ();
3     children   child_names_t := child_names_t ();
4 BEGIN
5     parents.EXTEND (2);
6     parents (1) := 'Samuel';
7     parents (2) := 'Charina';
8     --
9     children.EXTEND;
10    children (1) := 'Feather';
11
12    --
13    INSERT INTO family
14        ( surname, parent_names, children_names )
15        VALUES ( 'Assurty', parents, children );
16 END;
SQL> /

```

PL/SQL procedure successfully completed.

```

SQL> SELECT * FROM family
2 /

```

```

SURNAME
PARENT_NAMES
CHILDREN_NAMES
-----

```

```

Assurty
FIRST_NAMES_T('Samuel', 'Charina')
CHILD_NAMES_T('Feather')

```

| Строки | Описание |
|----------------------|---|
| Раздел А | Команда CREATE TYPE используется для создания двух типов VARRAY. Обратите внимание: с типом VARRAY необходимо задать максимальную длину коллекции. По сути мои объявления определяют некое подобие «социальной политики»: не более двух родителей и не более одного ребенка |
| Раздел В | Создание реляционной таблицы из трех столбцов: VARCHAR2 для фамилии и двух столбцов VARRAY (для родителей и детей) |
| Раздел С, строки 2–3 | Объявление двух локальных экземпляров VARRAY на основании типа уровня схемы. По аналогии с вложенными таблицами (и в отличие от ассоциативных массивов), мы должны вызвать функцию-конструктор, имя которой совпадает с именем TYPE, для инициализации структур |
| 5–10 | Расширение и заполнение коллекций; добавляются имена родителей и одного ребенка. При попытке добавления второго ребенка произойдет ошибка |
| 13–15 | Чтобы вставить строку в таблицу family, достаточно включить VARRAY в список значений. Как видите, в Oracle вставка коллекций в реляционную таблицу выполняется тривиально! |

Использование коллекций

В следующих разделах описываются особенности объявления и применения коллекций в разных местах программного кода. Поскольку типы коллекций могут определяться в базе данных (только вложенные таблицы и VARRAY), эти структуры можно задействовать не только в программах PL/SQL, но и в таблицах и объектных типах.

Коллекции как компоненты записи

Коллекции могут использоваться в записях наряду с другими типами данных. В состав записи может входить одна или несколько таких структур, включая массивы VARRAY, вложенные таблицы и ассоциативные массивы. Например:

```

CREATE OR REPLACE TYPE color_tab_t IS TABLE OF VARCHAR2(100)
/

```

продолжение ➤

```

DECLARE
  TYPE toy_rec_t IS RECORD (
    manufacturer INTEGER,
    shipping_weight_kg NUMBER,
    domestic_colors color_tab_t,
    international_colors color_tab_t
  );

```

Коллекции в качестве параметров программы

Коллекции также могут передаваться в параметрах функций и процедур. Формат объявления параметра остается неизменным (за дополнительной информацией обращайтесь к главе 17):

```

имя_параметра [ IN | IN OUT | OUT ] тип_параметра
[ [ NOT NULL ] [ DEFAULT | := значение_по_умолчанию ] ]

```

В PL/SQL нет обобщенных, заранее определенных типов коллекций (за исключением некоторых пакетов — таких, как DBMS_SQL и DBMS_UTILITY). Это означает, что для передачи коллекции в аргументе в программе должен быть определен соответствующий тип параметра. Существует несколько возможных способов:

- Определение типа на уровне схемы конструкцией CREATE TYPE.
- Объявление типа коллекции в пакете.
- Объявление типа из внешней области действия в определении модуля.

Пример использования типа уровня схемы:

```

CREATE TYPE yes_no_t IS TABLE OF CHAR(1);
/
CREATE OR REPLACE PROCEDURE act_on_flags (flags_in IN yes_no_t)
IS
BEGIN
  ...
END act_on_flags;
/

```

Пример использования типа коллекции, определяемого в спецификации пакета: существует только один способ объявления ассоциативного массива логических значений (и любых других базовых типов данных), так почему бы не определить их в спецификации пакета для последующих ссылок в приложениях?

```

/* Файл в Сети: aa_types.pks */
CREATE OR REPLACE PACKAGE aa_types
IS
  TYPE boolean_aat IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
  ...
END aa_types;
/

```

Внимание: тип коллекции в списке параметров должен уточняться именем пакета:

```

CREATE OR REPLACE PROCEDURE act_on_flags (
  flags_in IN aa_types.boolean_aat)
IS
BEGIN
  ...
END act_on_flags;
/

```

Наконец, в следующем примере тип коллекции объявляется во внешнем блоке, а затем используется во внутреннем блоке:

```

DECLARE
  TYPE birthdates_aat IS VARRAY (10) OF DATE;
  l_dates birthdates_aat := birthdates_aat ();

```

```

BEGIN
  l_dates.EXTEND (1);
  l_dates (1) := SYSDATE;
DECLARE
  FUNCTION earliest_birthdate (list_in IN birthdates_aat) RETURN DATE
  IS
  BEGIN
    ...
  END earliest_birthdate;
BEGIN
  DBMS_OUTPUT.put_line (earliest_birthdate (l_dates));
END;
END;

```

Коллекции как типы данных для значений, возвращаемых функцией

Ниже приведен пример функции, возвращающей значение `color_tab_t`. Этот же тип имеет и локальная переменная этой функции. В отношении возвращаемых функциями значений действует то же правило, что и в отношении их параметров: пользовательские типы данных должны быть определены вне модуля:

```

FUNCTION true_colors (whose_id IN NUMBER) RETURN color_tab_t
AS
  l_colors color_tab_t;
BEGIN
  SELECT favorite_colors INTO l_colors
  FROM personality_inventory
  WHERE person_id = whose_id;
  RETURN l_colors;
END;

```

(Конструкция `BULK COLLECT` описана в главе 15.) Как использовать эту функцию в программе PL/SQL? Поскольку она способна заменить переменную типа `color_type_t`, то вы можете либо присвоить возвращаемое ею значение переменной типа коллекции, либо присвоить переменной (тип которой совместим с типом элементов коллекции) один из элементов возвращаемой коллекции. Первый способ весьма прост. Однако обратите внимание, что это один из тех случаев, когда переменную коллекции не обязательно инициализировать явно:

```

DECLARE
  color_array color_tab_t;
BEGIN
  color_array := true_colors (8041);
END;

```

При втором способе сразу после вызова функции необходимо указать индекс требуемого элемента коллекции:

```

DECLARE
  one_of_my_favorite_colors VARCHAR2(30);
BEGIN
  one_of_my_favorite_colors := true_colors (8041) (1);
END;

```

Правда, при этом возникает одна проблема: если в таблице базы данных не окажется записи, в столбце `person_id` которой содержится значение 8041, то в ответ на попытку считать первый элемент получившейся коллекции будет инициировано исключение `COLLECTION_IS_NULL`. Его необходимо перехватить и обработать в соответствии с требованиями приложения.

Коллекции как «столбцы» таблицы базы данных

Вложенные таблицы и структуры `VARRAY` позволяют хранить в столбце таблицы базы данных неатомарные значения, записываемые или считываемые одной командой.

Например, в используемой отделом кадров таблице `employee` в одном из столбцов может храниться информация о датах рождения членов семьи сотрудников (табл. 12.1).

Таблица 12.1. Хранение списка дат в виде коллекции в столбце таблицы сотрудников

| Поле Id (NUMBER) | Поле Name (VARCHAR2) | Поле Dependents_ages (Dependent_birthdate_t) |
|---------------------|-------------------------|---|
| 10010 | Zaphod Beeblebrox | 12-JAN-1763 4-JUL-1977 22-MAR-2021 |
| 10020 | Molly Squiggly | 15-NOV-1968 15-NOV-1968 |
| 10030 | Joseph Josephs | |
| 10040 | Cepheus Usrbin | 27-JUN-1995 9-AUG-1996 19-JUN-1997 |
| 10050 | Deirdre Quattlebaum | 21-SEP-1997 |

Создать такую таблицу несложно. Сначала нужно определить тип коллекции:

```
CREATE TYPE Dependent_birthdate_t AS VARRAY(10) OF DATE;
```

который затем используется в определении таблицы:

```
CREATE TABLE employees (
  id NUMBER,
  name VARCHAR2(50),
  ...другие столбцы...,
  dependents_ages dependent_birthdate_t
);
```

Заполнить таблицу можно с помощью команды `INSERT`, в которой конструктор по умолчанию (см. далее в этой главе) преобразует список дат в значения нужного типа:

```
INSERT INTO employees VALUES (42, 'Zaphod Beeblebrox', ...,
  dependent_birthdate_t( '12-JAN-1765', '4-JUL-1977', '22-MAR-2021'));
```

Теперь рассмотрим пример использования столбца типа вложенной таблицы. Создавая внешнюю таблицу `personality_inventory`, Oracle необходимо указать имя таблицы для записи данных столбца типа вложенной таблицы:

```
CREATE TABLE personality_inventory (
  person_id NUMBER,
  favorite_colors color_tab_t,
  date_tested DATE,
  test_results BLOB)
NESTED TABLE favorite_colors STORE AS favorite_colors_st;
```

Конструкция `NESTED TABLE...STORE AS` сообщает Oracle, что таблица для хранения данных столбца `favorite_colors` (вспомогательная таблица) должна иметь имя `favorite_colors_st`. Эта таблица будет храниться отдельно от остальных данных таблицы `personality_inventory`. Ограничений на ее размер не существует.

Данные вспомогательной таблицы нельзя обрабатывать непосредственно. Попытка прямого считывания или записи в нее информации приведет к возникновению ошибки. Чтение или запись атрибутов этой таблицы возможны только через ссылку на внешнюю таблицу (псевдофункции коллекций рассматриваются далее в разделе «Работа с коллекциями в SQL»). Физические атрибуты вспомогательной таблицы также нельзя задать напрямую — они наследуются от «самой внешней» таблицы.

Главное различие между вложенными таблицами и структурами `VARRAY` проявляется при использовании их в качестве типов данных столбцов. Хотя массив `VARRAY`, подобно

вложенной таблице, позволяет сохранить в одном столбце множество значений, для него необходимо указать максимальную длину массива, который будет храниться в таблице вместе с остальными данными. Поэтому разработчики компании Oracle рекомендуют использовать столбцы типа `VARRAY` для «маленьких» массивов, а вложенные таблицы — для «больших».

Коллекции как атрибуты объектного типа

В следующем примере моделируются данные, используемые для определения характеристик автомобиля. Каждый объект типа `auto_spec_t` содержит набор цветов, в которые может быть покрашена машина.

```
CREATE TYPE auto_spec_t AS OBJECT (  
    make VARCHAR2(30),  
    model VARCHAR2(30),  
    available_colors color_tab_t  
);
```

Поскольку объектный тип не требует места для хранения данных, в его определении конструкцией `CREATE TYPE ... AS OBJECT` не нужно указывать имя вспомогательной таблицы. Оно задается позже, когда будем создавать таблицу со столбцом этого типа:

```
CREATE TABLE auto_specs OF auto_spec_t  
    NESTED TABLE available_colors STORE AS available_colors_st;
```

Приведенная выше инструкция требует пояснений. Создавая таблицу объектов, Oracle просматривает объявление объектного типа, чтобы определить, какие столбцы должны содержать такая таблица. Обнаружив, что один из столбцов (а именно `available_colors`) является вложенной таблицей, Oracle создает отдельную таблицу для хранения его данных. Предложение

```
...NESTED TABLE available_colors STORE AS available_colors_st
```

сообщает Oracle, что вспомогательная таблица для хранения данных столбца `available_colors` должна называться `available_colors_st`.

Подробнее об объектных типах Oracle рассказывается в главе 26.

Выбор типа коллекции

Какая разновидность коллекций лучше подходит для вашего приложения? В одних случаях выбор очевиден, в других возможны разные варианты. В этом разделе представлены некоторые рекомендации, которые помогут вам принять решение. В табл. 12.2 перечислены основные различия между ассоциативными массивами, вложенными таблицами и массивами `VARRAY`.

Как правило, разработчики PL/SQL инстинктивно склонны к использованию ассоциативных массивов. Почему? Потому что ассоциативные массивы требуют минимального объема кода. Их не нужно инициализировать или расширять. Традиционно они считались самой эффективной разновидностью коллекций (хотя со временем эти различия, вероятно, исчезнут). Но если коллекция должна храниться в таблице баз данных, ассоциативный массив отпадает. Остается вопрос: вложенная таблица или `VARRAY`?

Следующие рекомендации помогут вам принять решение. Впрочем, если к настоящему моменту еще не составили полного представления о коллекциях, лучше еще раз перечитать начало этой главы.

- Если вам нужна функциональность разреженных коллекций (например, для реализации «интеллектуального хранения данных»), единственным реальным вариантом остается ассоциативный массив. Да, вы можете выделять и удалять элементы при

работе с переменными вложенных таблиц (как показано далее при описании методов NEXT и PRIOR), но эти операции для коллекций сколько-нибудь значительного размера выполняются крайне неэффективно.

Таблица 12.2. Сравнение разновидностей коллекций в Oracle

| Характеристика | Ассоциативный массив | Вложенная таблица | Массив VARRAY |
|--|--|---|--|
| Размерность | Одномерная коллекция | Одномерная коллекция | Одномерная коллекция |
| Может использоваться в SQL? | Нет | Да | Да |
| Может использоваться как тип данных столбца таблицы? | Нет | Да; данные хранятся в отдельной вспомогательной таблице | Да; данные хранятся в той же таблице |
| Неинициализированное состояние | Пустая коллекция (не может быть равна NULL); элементы не определены | Атомарное значение NULL; ссылки на элементы недействительны | Атомарное значение NULL; ссылки на элементы недействительны |
| Инициализация | Автоматическая при объявлении | При вызове конструктора, выборке или присваивании | При вызове конструктора, выборке или присваивании |
| Ссылка в элементах PL/SQL | BINARY_INTEGER и любые из его подтипов (-2 147 483 647 .. 2,147,483,647) | VARCHAR2 (Oracle9i Release2 и выше) | Положительное целое число от 1 до 2 147 483 647 |
| Разрезанная? | Да | Изначально нет, но может стать после удалений | Нет |
| Ограниченная? | Нет | Может расширяться | Да |
| Допускает присваивание любому элементу в любой момент времени? | Да | Нет; может потребоваться предварительный вызов EXTEND | Нет; может потребоваться предварительный вызов EXTEND, который не может выйти за верхнюю границу |
| Способ расширения | Присваивание значения элементу с новым индексом | Встроенная процедура EXTEND (или TRIM для сжатия) без заранее определенного максимума | EXTEND (или TRIM), но не более объявленного максимального размера |
| Поддерживает проверку равенства? | Нет | Да, Oracle10g и выше | Нет |
| Поддерживает использование операций над множествами? | Нет | Да, Oracle10g и выше | Нет |
| Сохраняет порядок следования элементов и индексы при сохранении в базе данных? | – | Нет | Да |

- Если приложение PL/SQL требует использования отрицательных индексов, придется использовать ассоциативные массивы.
- Если вы работаете в Oracle10g или более поздней версии и с коллекциями выполняются высокоуровневые операции из теории множеств, используйте вложенные таблицы вместо ассоциативных массивов.
- Если количество строк, хранящихся в таблице, должно быть жестко ограничено, используйте массивы VARRAY.

- Если в столбцовой коллекции должны храниться большие объемы постоянных данных, единственным реальным вариантом остается вложенная таблица. База данных будет использовать отдельную скрытую таблицу для хранения данных коллекции, что обеспечивает ее почти неограниченный рост.
- Если вы хотите сохранить порядок элементов, хранимых в столбце коллекции и набор данных будет относительно небольшим, используйте массив `VARRAY`.
- Что считать «относительно небольшим»? Ориентируйтесь на объем данных, помещающихся в одном блоке базы данных; при выходе за пределы блока начинаются переходы между записями, снижающие производительность операций.
- Несколько признаков ситуации, в которой предпочтение отдается массиву `VARRAY`: вам не нужно беспокоиться о возможных удалениях в середине набора данных; сама природа данных подразумевает наличие четкой верхней границы; в большинстве случаев при выборке извлекается все содержимое коллекции.

Встроенные методы коллекций

PL/SQL предоставляет для создаваемых вами коллекций множество встроенных функций и процедур, называемых методами коллекций. Эти методы предназначены для получения информации о содержимом коллекции и ее изменения. Их полный список приведен в табл. 12.3.

Таблица 12.3. Методы коллекций

| Метод (функция или процедура) | Описание |
|-------------------------------|--|
| COUNT (функция) | Возвращает текущее значение элементов в коллекции |
| DELETE (процедура) | Удаляет из коллекции один или несколько элементов. Уменьшает значение, возвращаемое функцией COUNT, если заданные элементы еще не удалены. Со структурами <code>VARRAY</code> может использоваться только для удаления всего содержимого |
| EXISTS (функция) | Возвращает значение TRUE или FALSE, определяющее, существует ли в коллекции заданный элемент |
| EXTEND (процедура) | Увеличивает количество элементов во вложенной таблице или <code>VARRAY</code> , а также значение, возвращаемое функцией COUNT |
| FIRST, LAST (функции) | Возвращают индексы первого (FIRST) и последнего (LAST) элемента в коллекции |
| LIMIT (функция) | Возвращает максимальное количество элементов в массиве <code>VARRAY</code> |
| PRIOR, NEXT (функции) | Возвращают индексы элементов, предшествующих заданному (PRIOR) и следующему за ним (NEXT). Всегда используйте PRIOR и NEXT для перебора коллекций, особенно при работе с разреженными (или потенциально разреженными) коллекциями |
| TRIM (функция) | Удаляет элементы, начиная с конца коллекции (элемент с наибольшим индексом) |

Все эти конструкции называются *методами*, потому что синтаксис их вызова отличается от синтаксиса вызова обычных процедур и функций. Это типичный синтаксис вызова методов, используемый в объектно-ориентированных языках программирования — таких, как Java.

Рассмотрим синтаксис вызова методов на примере LAST. Эта функция возвращает наибольший индекс элемента ассоциативного массива. Стандартный вызов этой функции выглядел бы так:

```
IF LAST (company_table) > 10 THEN ... /* Неверный синтаксис */
```

Иначе говоря, ассоциативный массив передается ей в качестве аргумента. Но поскольку функция LAST является методом, она «принадлежит» объекту — в данном случае ассоциативному массиву. Правильный синтаксис ее вызова выглядит так:

```
IF company_table.LAST > 10 THEN ... /* Правильный синтаксис */
```

В общем случае синтаксис вызова методов ассоциативного массива выглядит так:

- Операция, не требующая передачи аргументов:

```
имя_таблицы.операция
```

- Операция, аргументами которой являются индексы элементов:

```
имя_таблицы.операция(индекс [, индекс])
```

Например, следующая команда возвращает TRUE, если в ассоциативном массиве `company_tab` определена запись 15:

```
company_tab.EXISTS(15)
```

Методы коллекций недоступны из SQL; их можно использовать только в программах PL/SQL.

Метод COUNT

Метод COUNT возвращает количество элементов в ассоциативном массиве, вложенной таблице или массиве VARRAY. В значении не учитываются элементы, удаленные из коллекции методом DELETE или TRIM.

Синтаксис вызова:

```
FUNCTION COUNT RETURN PLS_INTEGER;
```

Рассмотрим пример. Прежде чем что-либо делать с коллекцией, мы проверяем, содержит ли она хотя бы один элемент:

```
DECLARE
    volunteer_list volunteer_list_ar := volunteer_list_ar('Steven');
BEGIN
    IF volunteer_list.COUNT > 0
    THEN
        assign_tasks (volunteer_list);
    END IF;
END;
```

Граничные условия

Для инициализированной коллекции, не содержащей ни одного элемента, COUNT возвращает нуль. Это же значение возвращается при вызове COUNT для пустого ассоциативного массива.

Возможные исключения

При вызове метода COUNT для неинициализированной вложенной таблицы или VARRAY инициируется заранее определенное исключение COLLECTION_IS_NULL. Такое исключение не может возникнуть при работе с ассоциативными массивами, не требующими инициализации.

Метод DELETE

Метод DELETE предназначен для удаления одного, нескольких или всех элементов ассоциативного массива, вложенной таблицы или массива VARRAY. При вызове без аргументов он удаляет все элементы коллекции. Вызов DELETE(i) удаляет i-й элемент вложенной

таблицы или ассоциативного массива. А вызов `DELETE(i, j)` удаляет все элементы с индексами от `i` до `j` включительно. Если коллекция представляет собой ассоциативный массив, индексируемый строками, `i` и `j` должны быть строковыми значениями; в противном случае они являются целыми числами.

При вызове с аргументами метод резервирует место, занимавшееся «удаленным» элементом, и позднее этому элементу можно присвоить новое значение.

Фактически PL/SQL освобождает память лишь при условии, что программа удаляет количество элементов, достаточное для освобождения *целой страницы памяти*. (Если же метод `DELETE` вызывается без параметров и очищает всю коллекцию, память освобождается немедленно.)



Применительно к массивам `VARRAY` метод `DELETE` может вызываться только без аргументов. Иначе говоря, с помощью указанного метода из этой структуры нельзя удалять отдельные элементы, поскольку в таком случае она станет разреженной, что недопустимо. Единственный способ удалить из `VARRAY` один или несколько элементов — воспользоваться методом `TRIM`, предназначенным для удаления группы расположенных рядом элементов, начиная с конца коллекции.

Следующая процедура удаляет из коллекции все элементы, кроме последнего. В ней используются четыре метода: `FIRST` — для получения номера первого удаляемого элемента; `LAST` — для получения номера последнего удаляемого элемента; `PRIOR` — для определения номера предпоследнего элемента; `DELETE` — для удаления всех элементов, кроме последнего:

```
PROCEDURE keep_last (the_list IN OUT List_t)
AS
    first_elt PLS_INTEGER := the_list.FIRST;
    next_to_last_elt PLS_INTEGER := the_list.PRIOR(the_list.LAST);
BEGIN
    the_list.DELETE(first_elt, next_to_last_elt);
END;
```

Несколько дополнительных примеров:

- Удаление всех строк из таблицы `names`:
`names.DELETE;`
- Удаление 77-й строки из таблицы `globals`:
`globals.DELETE (77);`
- Удаление из таблицы `temp_reading` всех элементов, начиная с индекса `-15 000` и до индекса `0` включительно:
`temp_readings.DELETE (-15000, 0);`

Граничные условия

Если значения индексов `i` и/или `j` указывают на несуществующие элементы, `DELETE` пытается «сделать наилучшее» и не генерирует исключение. Например, если таблица содержит три элемента с индексами 1, 2 и 3, то вызов метода `DELETE(-5, 1)` удалит только один элемент с индексом 1, а вызов `DELETE(-5)` не изменит состояния коллекции.

Возможные исключения

Вызов метода `DELETE` для неинициализированной вложенной таблицы или массива `VARRAY` инициирует исключение `COLLECTION_IS_NULL`.

Метод EXISTS

Метод EXISTS используется с вложенными таблицами, ассоциативными массивами и массивами VARRAY для определения наличия в коллекции заданного элемента. Если таковой имеется, метод возвращает значение TRUE, а если отсутствует — значение FALSE. Значение NULL не возвращается ни при каких условиях. Кроме того, EXISTS возвращает FALSE и в том случае, если заданный элемент был удален из коллекции с помощью метода TRIM или DELETE.

Следующий блок проверяет, присутствует ли заданный элемент в коллекции, и при положительном ответе присваивает ему значение NULL:

```
DECLARE
    my_list color_tab_t := color_tab_t();
    element PLS_INTEGER := 1;
BEGIN
    ...
    IF my_list.EXISTS(element)
    THEN
        my_list(element) := NULL;
    END IF;
END;
```

Граничные условия

Если метод EXISTS вызывается для неинициализированной (содержащей атомарное значение NULL) вложенной таблицы или структуры VARRAY либо для инициализированной коллекции, не содержащей ни одного элемента, он просто возвращает значение FALSE. Поэтому его можно вызывать без предварительной проверки вызовом COUNT, не рискуя получить сообщение об ошибке.

Возможные исключения

Метод EXISTS не инициирует исключения.

Метод EXTEND

Чтобы добавить элемент во вложенную таблицу или массив VARRAY, необходимо сначала выделить для него область памяти. Соответствующая операция, не зависящая от присваивания элементу значения, выполняется методом EXTEND. Следует помнить, что указанный метод неприменим к ассоциативным массивам.

Метод EXTEND, как уже было сказано, добавляет элементы в коллекцию. При вызове без аргументов он добавляет один элемент со значением NULL. Вызов EXTEND(n) присоединяет n элементов со значением NULL, а вызов EXTEND(n, i) — n элементов, и всем им присваивает значение i-го элемента. Последняя форма метода применяется к коллекциям, для элементов которых задано ограничение NOT NULL.

Синтаксис перегруженного метода EXTEND:

```
PROCEDURE EXTEND (n PLS_INTEGER:=1);
PROCEDURE EXTEND (n PLS_INTEGER, i PLS_INTEGER);
```

В следующем примере процедура push добавляет в список один элемент и присваивает ему новое значение:

```
PROCEDURE push (the_list IN OUT List_t, new_value IN VARCHAR2)
AS
BEGIN
    the_list.EXTEND;
    the_list(the_list.LAST) := new_value;
END;
```

В другом фрагменте кода метод `EXTEND` используется для включения в коллекцию 10 новых элементов с одинаковыми значениями. Сначала в коллекцию добавляется один элемент, которому явно присваивается нужное значение. При повторном вызове метода `EXTEND` в коллекцию добавляется еще 9 элементов, которым присваивается значение первого элемента коллекции `new_value`:

```
PROCEDURE push_ten (the_list IN OUT List_t, new_value IN VARCHAR2)
AS
    l_copyfrom PLS_INTEGER;
BEGIN
    the_list.EXTEND;
    l_copyfrom := the_list.LAST;
    the_list(l_copyfrom) := new_value;
    the_list.EXTEND (9, l_copyfrom);
END;
```

Граничные условия

Если параметр `n` имеет значение `NULL`, метод не выполнит никаких действий.

Возможные исключения

При вызове метода `EXTEND` для неинициализированной вложенной таблицы или `VARRAY` инициируется исключение `COLLECTION_IS_NULL`. Попытка добавить в массив `VARRAY` элементы, индекс которых превышает максимальный индекс массива в его объявлении, инициирует исключение `SUBSCRIPT_BEYOND_LIMIT`.

Методы `FIRST` и `LAST`

Методы `FIRST` и `LAST` возвращают соответственно наименьший и наибольший индексы элементов вложенной таблицы, ассоциативного массива или массива `VARRAY`. Для ассоциативных массивов, индексируемых строками, эти методы возвращают строки; «наименьшее» и «наибольшее» значения определяются порядком набора символов, используемого данным сеансом. Для других типов коллекций методы возвращают целые числа.

Синтаксис этих функций:

```
FUNCTION FIRST RETURN PLS_INTEGER | VARCHAR2;
FUNCTION LAST RETURN PLS_INTEGER | VARCHAR2;
```

Так, следующий фрагмент перебирает все элементы коллекции от начала к концу:

```
FOR indx IN holidays.FIRST .. holidays.LAST
LOOP
    send_everyone_home (indx);
END LOOP;
```

Запомните, что такой цикл будет выполнен корректно (то есть не породит исключения `NO_DATA_FOUND`) лишь при условии, что коллекция является плотной.

В следующем примере для добавления элементов в конец ассоциативного массива используется оператор `COUNT`. Цикл `FOR` с курсором используется для копирования данных из базы в ассоциативный массив. При выборке первой записи коллекция `companies` пуста, поэтому `COUNT` возвращает 0.

```
FOR company_rec IN company_cur
LOOP
    companies ((companies.COUNT) + 1).company_id
        company_rec.company_id;
END LOOP;
```

Граничные условия

Если методы `FIRST` и `LAST` вызываются для инициализированных коллекций, не содержащих ни одного элемента, они возвращают `NULL`. Для массива `VARRAY`, всегда содержащего хотя бы один элемент, `FIRST` всегда возвращает 1, а `LAST` — то же значение, что и метод `COUNT`.

Возможные исключения

При вызове методов `FIRST` и `LAST` для неинициализированной вложенной таблицы или массива `VARRAY` инициируется исключение `COLLECTION_IS_NULL`.

Метод LIMIT

Метод `LIMIT` возвращает максимальное количество элементов, которое можно определить в массиве `VARRAY`. В случае вложенной таблицы или ассоциативного массива он возвращает `NULL`. Синтаксис этого метода:

```
FUNCTION LIMIT RETURN PLS_INTEGER;
```

В следующем примере перед добавлением нового элемента в конец массива `VARRAY` мы сначала проверяем, есть ли в нем еще свободное место:

```
IF my_list.LAST < my_list.LIMIT
THEN
    my_list.EXTEND;
END IF;
```

Граничные условия

У метода `LIMIT` граничных условий не существует.

Возможные исключения

Вызов метода `LIMIT` для неинициализированной вложенной таблицы или массива `VARRAY` генерирует исключение `COLLECTION_IS_NULL`.

Методы PRIOR и NEXT

Методы `PRIOR` и `NEXT` используются для перемещения по коллекциям — вложенным таблицам, ассоциативным массивам и массивам `VARRAY`. Метод `PRIOR` возвращает индекс предыдущего, а метод `NEXT` — следующего элемента коллекции. Следующая функция возвращает сумму чисел, хранящихся в коллекции `list_t`:

```
FUNCTION compute_sum (the_list IN list_t) RETURN NUMBER
AS
    row_index PLS_INTEGER := the_list.FIRST;
    total NUMBER := 0;
BEGIN
    LOOP
        EXIT WHEN row_index IS NULL;
        total := total + the_list(row_index);
        row_index := the_list.NEXT(row_index);
    END LOOP;
    RETURN total;
END compute_sum;
```

Та же программа, но с перебором элементов от последней к первой определенной записи коллекции:

```
FUNCTION compute_sum (the_list IN list_t) RETURN NUMBER
AS
    row_index PLS_INTEGER := the_list.LAST;
    total NUMBER := 0;
```

```

BEGIN
  LOOP
    EXIT WHEN row_index IS NULL;
    total := total + the_list(row_index);
    row_index := the_list.PRIOR(row_index);
  END LOOP;
  RETURN total;
END compute_sum;

```

В данном случае направление перемещения не имеет значения, но в других программах оно может повлиять на результат обработки.

Граничные условия

Методы `PRIOR` и `NEXT` для инициализированной коллекции, не содержащей ни одного элемента, возвращают `NULL`. Если значение `i` больше или равно `COUNT`, метод `NEXT` возвращает `NULL`; если `i` меньше или равно `FIRST`, метод `PRIOR` возвращает `NULL`.



Вплоть до версии Oracle12c, если коллекция не пуста, а параметр `i` больше или равен `COUNT`, метод `PRIOR` возвращает `LAST`; если параметр `i` меньше `FIRST`, метод `NEXT` возвращает `FIRST`. Однако сохранение такого поведения в будущих версиях Oracle не гарантировано.

Возможные исключения

Вызов методов `PRIOR` и `NEXT` для неинициализированной вложенной таблицы или массива `VARRAY` генерирует исключение `COLLECTION_IS_NULL`.

Метод TRIM

Метод `TRIM` удаляет `n` последних элементов коллекции — вложенной таблицы или массива `VARRAY`. Если метод вызывается без аргументов, он удалит только один элемент. Как упоминалось ранее, при совместном использовании методов `TRIM` и `DELETE` возможна накладка: если заданный в вызове метода `TRIM` элемент был уже удален методом `DELETE`, метод `TRIM` «повторит» удаление, но считает его частью `n`, поэтому количество реально удаленных элементов окажется меньшим, чем вы рассчитывали.



Попытка вызова метода `TRIM` для ассоциативного массива приведет к ошибке компиляции.

Синтаксис метода `TRIM`:

```
PROCEDURE TRIM (n PLS_INTEGER:=1);
```

Следующая функция извлекает из списка последнее значение и возвращает его вызывающему блоку. Операция извлечения реализуется как выборка значения с последующим усечением коллекции на один элемент:

```

FUNCTION pop (the_list IN OUT list_t) RETURN VARCHAR2
AS
  l_value VARCHAR2(30);
BEGIN
  IF the_list.COUNT >= 1
  THEN
    /* Сохраняем значение последнего элемента коллекции,
    || которое будет возвращено функцией

```

```
*/  
l_value := the_list(the_list.LAST);  
the_list.TRIM;  
END IF;  
RETURN l_value;  
END;
```

Граничные условия

Если значение `n` равно `NULL`, метод не выполнит никаких действий.

Возможные исключения

При попытке удалить больше элементов, чем имеется в коллекции, инициируется исключение `SUBSCRIPT_BEYOND_COUNT`. Если метод `TRIM` вызывается для неинициализированной вложенной таблицы или массива `VARRAY`, инициируется исключение `COLLECTION_IS_NULL`.



Вызывая методы `TRIM` и `DELETE` для одной и той же коллекции, можно получить неожиданные результаты. На сколько элементов станет меньше в коллекции, если удалить последний элемент методом `DELETE`, а затем вызвать метод `TRIM` с тем же значением параметра? Казалось бы, это приведет к удалению двух элементов, но в действительности оба метода удалят один и тот же элемент. Чтобы избежать накладок, компания Oracle рекомендует использовать только один из этих двух методов при работе с конкретной коллекцией.

Работа с коллекциями

Итак, мы познакомились с основными разновидностями и методами коллекций. Мы рассмотрели примеры работы с ассоциативными массивами, вложенными таблицами и массивами `VARRAY`. Пора переходить к подробному рассмотрению практической работы с коллекциями в программах. В этом разделе рассматриваются следующие вопросы:

- Обработка исключений при работе с коллекциями.
- Объявление типов коллекций.
- Объявление и инициализация переменных коллекций.
- Присваивание значений коллекциям.
- Использование коллекций составных типов данных (например, коллекций, элементы которых представляют собой коллекции).
- Работа с последовательными и непоследовательными ассоциативными массивами.
- Возможности коллекций со строковым индексированием.
- Работа с коллекциями `PL/SQL` в командах `SQL`.

Объявление типов коллекций

Прежде чем работать с коллекцией, ее необходимо объявить, причем объявление должно базироваться на типе коллекции. Таким образом, прежде всего необходимо научиться определять типы коллекций.

Существует два способа определения типов коллекций:

- Тип коллекции можно объявить в программе `PL/SQL` с применением синтаксиса `TYPE`. Такой тип будет доступен лишь в том блоке, где он был объявлен. Тип, определяемый в спецификации пакета, будет доступен всем программам, схема которых обладает правами `EXECUTE` для данного пакета.

- Определение типа вложенной таблицы или VARRAY как объекта уровня схемы в базе данных Oracle с использованием команды CREATE TYPE. В дальнейшем такой тип может использоваться в качестве типа столбцов таблиц баз данных и атрибутах объектных типов, а также для объявления переменных в программах PL/SQL. Он может использоваться в любой программе схемы, обладающей соответствующими правами EXECUTE.

Объявление ассоциативного массива

Конструкция TYPE для определения ассоциативного массива имеет следующий формат:

```
TYPE имя_типа_таблицы IS TABLE OF тип_данных [ NOT NULL ]
INDEX BY тип_индекса;
```

Здесь *имя_типа_таблицы* — имя создаваемой коллекции, *тип_данных* — тип данных единственного столбца коллекции, а *тип_индекса* — тип данных индекса, используемого для упорядочения содержимого коллекции. При желании можно добавить ограничение NOT NULL, указывающее, что каждая строка таблицы обязательно должна содержать значение.

Имена табличных типов подчиняются тем же правилам, что и имена других идентификаторов PL/SQL: имя должно начинаться с буквы, иметь длину до 30 символов и содержать некоторые специальные символы (решетка, подчеркивание, знак доллара). А если заключить имя в двойные кавычки, оно может содержать до 30 *любых* символов.

Типом данных элементов коллекции может быть *почти* любой тип данных PL/SQL. Допускается использование большинства скалярных базовых типов данных, подтипов, типов с привязкой и типов, определяемых пользователем. Исключения составляют типы REF CURSOR (нельзя создать коллекцию курсорных переменных) и исключения.

Тип_индекса определяет тип данных, которые определяют местонахождение данных, размещаемых в коллекции. До Oracle9i Release 2 поддерживался только один способ определения индекса ассоциативного массива:

```
INDEX BY PLS_INTEGER
```

Теперь тип данных индекса может определяться и по-другому:

```
INDEX BY BINARY_INTEGER
INDEX BY PLS_INTEGER
INDEX BY POSITIVE
INDEX BY NATURAL
INDEX BY SIGNED_INTEGER /* Только три значения индекса: 1, 0 and 1 ! */
INDEX BY VARCHAR2(32767)
INDEX BY таблица.столбец%TYPE
INDEX BY курсор.столбец%TYPE
INDEX BY пакет.переменная%TYPE
INDEX BY пакет.подтип
```

Несколько примеров объявлений типов ассоциативных массивов:

```
-- Список дат
TYPE birthdays_tt IS TABLE OF DATE INDEX BY PLS_INTEGER;

-- Список идентификаторов компаний
TYPE company_keys_tt IS TABLE OF company.company_id%TYPE NOT NULL
INDEX BY PLS_INTEGER;

-- Список записей книг; эта структура позволяет создать
-- "локальную" копию таблицы книг в программе PL/SQL.
TYPE booklist_tt IS TABLE OF books%ROWTYPE
INDEX BY NATURAL;
```

```
-- Каждая коллекция упорядочивается по имени автора.
TYPE books_by_author_tt IS TABLE OF books%ROWTYPE
  INDEX BY books.author%TYPE;

-- Коллекция коллекций
TYPE private_collection_tt IS TABLE OF books_by_author_tt
  INDEX BY VARCHAR2(100);
```

В предыдущем примере я объявил обобщенный тип коллекции (список дат), но присвоил ему конкретное имя: `birthdays_tt`. Конечно, это всего лишь один из способов объявления типа ассоциативных массивов дат. Вместо того чтобы создавать набор определений `TYPE` коллекций, различающихся только по имени и разбросанных по всему приложению, стоит рассмотреть возможность создания одного пакета с набором заранее определенных, стандартных типов коллекций. Следующий пример находится в файле `colltypes.pks` на сайте книги:

```
/* Файл в Сети: colltypes.pks */
PACKAGE collection_types
IS
  -- Типы ассоциативных массивов
  TYPE boolean_aat IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
  TYPE date_aat IS TABLE OF DATE INDEX BY PLS_INTEGER;
  TYPE pls_integer_aat IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
  TYPE number_aat IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  TYPE identifier_aat IS TABLE OF VARCHAR2(30)
    INDEX BY PLS_INTEGER;
  TYPE vcmay_aat IS TABLE OF VARCHAR2(32767)
    INDEX BY PLS_INTEGER;

  -- Типы вложенных таблиц
  TYPE boolean_ntt IS TABLE OF BOOLEAN;
  TYPE date_ntt IS TABLE OF DATE;
  TYPE pls_integer_ntt IS TABLE OF PLS_INTEGER;
  TYPE number_ntt IS TABLE OF NUMBER;
  TYPE identifier_ntt IS TABLE OF VARCHAR2(30);
  TYPE vcmay_ntt IS TABLE OF VARCHAR2(32767);
END collection_types;
/
```

При наличии такого пакета можно предоставить полномочия `EXECUTE` группе `PUBLIC`, и тогда все разработчики смогут использовать пакетные определения `TYPE` для объявления своих коллекций. Пример:

```
DECLARE
  family_birthdays collection_types.date_aat;
```

Объявление вложенной таблицы или VARRAY

Как и в случае с ассоциативными массивами, перед объявлением вложенной таблицы или `VARRAY` также необходимо определить тип. Эти типы определяются либо в базе данных, либо в блоке `PL/SQL`.

Тип данных вложенной таблицы, которая существует в базе данных (а не только в коде `PL/SQL`), создается следующим образом:

```
CREATE [ OR REPLACE ] TYPE имя_типа AS | IS
  TABLE OF тип_элемента [ NOT NULL ];
```

Тип данных `VARRAY`, который существует в базе данных (а не только в коде `PL/SQL`), создается следующим образом:

```
CREATE [ OR REPLACE ] TYPE имя_типа AS | IS
  VARRAY (максимальный_индекс) OF тип_элемента [ NOT NULL ];
```


Удаление типа осуществляется следующей командой:

```
DROP TYPE имя_типа [ FORCE ];
```

А вот как в программе PL/SQL создается тип вложенной таблицы:

```
TYPE имя_типа IS TABLE OF тип_элемента [ NOT NULL ];
```

Для объявления типа данных VARRAY в PL/SQL используется объявление:

```
TYPE имя_типа IS VARRAY (максимальный_индекс)  
OF тип_элемента [ NOT NULL ];
```

Основные параметры, используемые при определении типов:

- Опция OR REPLACE позволяет переопределить существующий тип данных с сохранением всех существующих привилегий (вместо удаления и повторного создания типа).
- Параметр *имя_типа* — допустимый идентификатор SQL или PL/SQL, который позднее будет использоваться в объявлениях переменных и столбцов.
- Параметр *тип_элемента* определяет тип данных элементов коллекции. Все элементы относятся к одному типу, которым может быть почти любой скалярный или объектный (в том числе REF) тип данных. При определении типа на уровне схемы тип данных должен быть типом данных SQL (логические значения недопустимы!).
- Опция NOT NULL указывает, что переменная данного типа не должна содержать элементов NULL. При этом вся коллекция может быть равна NULL (не инициализирована).
- Параметр *максимальный* — значение, определяющее максимальное количество элементов в массиве VARRAY.
- Опция FORCE приказывает Oracle удалить тип данных, даже если на него ссылается другой тип. Например, если в определении объектного типа указывается тип коллекции, его можно удалить, добавив в соответствующую команду ключевое слово FORCE.



За инструкцией CREATE TYPE должен следовать символ косой черты (/), как при создании процедуры, функции или пакета.

Обратите внимание: единственным различием в синтаксисе определения типов вложенной таблицы и ассоциативного массива является отсутствие в первом случае предложения INDEX BY.

Синтаксис определения массива VARRAY отличается от синтаксиса определения типа вложенной таблицы наличием ключевого слова VARRAY и ограничением количества элементов.

Изменение характеристик вложенных таблиц и массивов VARRAY

Инструкция ALTER TYPE позволяет изменить некоторые характеристики вложенной таблицы или типа VARRAY, созданных в базе данных. Количество элементов типа VARRAY увеличивается командой ALTER TYPE ... MODIFY LIMIT:

```
ALTER TYPE list_vat MODIFY LIMIT 100 INVALIDATE;  
/
```

Если элемент типа VARRAY или вложенной таблицы относится к символьному типу переменной длины, RAW или числовому типу, вы можете увеличить его размер или точность. Пример:

```
CREATE TYPE list_vat AS VARRAY(10) OF VARCHAR2(80);  
/  
ALTER TYPE list_vat MODIFY ELEMENT TYPE VARCHAR2(100) CASCADE;  
/
```

Опции `INVALIDATE` и `CASCADE` соответственно объявляют недействительными все зависимые объекты и распространяют внесенные изменения как в зависимых типах, так и в таблицах.

Объявление и инициализация переменных-коллекций

Созданный тип коллекции указывается при объявлении экземпляра этого типа, то есть переменной. Общий формат объявления коллекции выглядит так:

```
имя_коллекции тип_коллекции [:= тип_коллекции (...)];
```

Здесь *имя_коллекции* — имя переменной-коллекции, а *тип_коллекции* — имя ранее объявленного типа и (для вложенных таблиц и `VARRAY`) одноименной функции-конструктора.

Имя конструктора совпадает с именем типа. При вызове конструктора передаются аргументы — список элементов, разделенных запятыми. Вложенные таблицы и массивы `VARRAY` должны инициализироваться перед использованием. В противном случае будет выдано следующее сообщение об ошибке:

```
ORA-06531: Reference to uninitialized collection
```

В следующем примере определяется общий тип коллекции, повторяющий структуру таблицы `company`. Затем на базе этого типа объявляются две переменные:

```
DECLARE
    TYPE company_aat IS TABLE OF company%ROWTYPE INDEX BY PLS_INTEGER;
    premier_sponsor_list company_aat;
    select_sponsor_list company_aat;
BEGIN
    ...
END;
```

Вложенную таблицу или массив `VARRAY` можно немедленно инициализировать при объявлении вызовом функции-конструктора. Пример:

```
DECLARE
    TYPE company_aat IS TABLE OF company%ROWTYPE;
    premier_sponsor_list company_aat := company_aat();
BEGIN
    ...
END;
```

Вложенная таблица также может инициализироваться в исполняемом разделе:

```
DECLARE
    TYPE company_aat IS TABLE OF company%ROWTYPE;
    premier_sponsor_list company_aat;
BEGIN
    premier_sponsor_list := company_aat();
END;
```

Главное — не забудьте инициализировать коллекцию перед ее использованием. Ассоциативные массивы не нужно (да и невозможно) инициализировать перед присваиванием значений. Как видите, объявление переменных-коллекций (или экземпляров типов коллекций) ничем не отличается от объявления других переменных: вы точно так же задаете имя, тип и необязательное исходное значение.

Давайте поближе познакомимся с инициализацией вложенных таблиц и `VARRAY`. Предыдущий пример показывает, как инициализировать коллекцию вызовом функции-конструктора без параметров. Также можно указать исходный набор значений. Предположим, я создаю тип уровня схемы с именем `color_tab_t`:

```
CREATE OR REPLACE TYPE color_tab_t AS TABLE OF VARCHAR2(30)
```

после чего объявляю переменные PL/SQL на основании этого типа:

```
DECLARE
  my_favorite_colors color_tab_t := color_tab_t();
  his_favorite_colors color_tab_t := color_tab_t('PURPLE');
  her_favorite_colors color_tab_t := color_tab_t('PURPLE', 'GREEN');
```

В первом объявлении коллекция инициализируется как пустая; она не содержит строк. Второе объявление присваивает единственное значение PURPLE строке 1 вложенной таблицы. Третье объявление присваивает два значения, PURPLE и GREEN, строкам 1 и 2 вложенной таблицы.

Так как я не присвоил никаких значений `my_favorite_colors` при вызове конструктора, я должен расширить коллекцию перед размещением элементов. Коллекции `his` и `her` уже были неявно расширены в соответствии со списком значений, переданным конструктору. Присваивание через функцию-конструктор подчиняется тем же ограничениям, которые действуют при прямом присваивании. Если, например, массив `VARRAY` ограничивается пятью элементами, а вы пытаетесь инициализировать его конструктором с шестью элементами, база данных выдаст ошибку *ORA-06532: Subscript outside of limit error*.

Неявная инициализация путем непосредственного присваивания

Если две коллекции относятся к одному типу данных, все содержимое одной коллекции можно скопировать в другую коллекцию (два разных типа коллекций, базирующиеся на одном типе данных, *не подойдут*). При этом инициализация выполняется автоматически. Следующий пример демонстрирует неявное присваивание, выполняемое при присваивании переменной `wedding_colors` значения переменной `earth_colors`.

```
DECLARE
  earth_colors color_tab_t := color_tab_t ('BRICK', 'RUST', 'DIRT');
  wedding_colors color_tab_t;
BEGIN
  wedding_colors := earth_colors;
  wedding_colors(3) := 'CANVAS';
END;
```

Этот код инициализирует переменную `wedding_colors` и создает в ней три элемента, соответствующие элементам `earth_colors`. В результате мы получаем две независимые переменные с одинаковыми значениями (в отличие от указателей на идентичные значения); скажем, если присвоить третьему элементу `wedding_colors` значение CANVAS, третий элемент коллекции `earth_colors` останется неизменным.

Учтите, что простой «совместимости типов данных» для этого недостаточно. Даже если объявить два типа коллекций с одинаковыми определениями, вы получите два разных типа данных, причем переменные одного типа нельзя будет присвоить переменным другого типа. Таким образом, следующий блок кода не будет компилироваться:

```
DECLARE
  TYPE tt1 IS TABLE OF employees%ROWTYPE;
  TYPE tt2 IS TABLE OF employees%ROWTYPE;
  t1  tt1 := tt1();
  t2  tt2 := tt2();
BEGIN
  /* Ошибка "PLS-00382: expression is of wrong type" */
  t1 := t2;
END;
```

Неявная инициализация путем выборки

Если коллекция используется в качестве типа в таблице базы данных, Oracle предоставляет очень элегантный способ перемещения данных коллекции между таблицей и кодом PL/SQL. Как и при прямом присваивании, при выборке данных из таблицы

и присваивании их переменной, объявленной как коллекция, происходит автоматическая инициализация этой переменной. Коллекции порой оказываются невероятно полезными! Давайте повнимательнее разберемся, как с помощью одной команды считать из базы данных все содержимое коллекции. Прежде всего создается таблица с коллекцией, в которую записывается несколько значений:

```
CREATE TABLE color_models (
    model_type VARCHAR2(12)
    , colors color_tab_t
)
NESTED TABLE colors STORE AS color_model_colors_tab
/

BEGIN
    INSERT INTO color_models
    VALUES ('RGB', color_tab_t ('RED', 'GREEN', 'BLUE'));
END;
/
```

А теперь — самое главное. За один цикл обращения к базе данных мы считываем все значения из столбца `colors` заданной строки и помещаем их в локальную переменную:

```
DECLARE
    l_colors color_tab_t;
BEGIN
    /* Выборка всех вложенных значений за одну операцию выборки.
    || Это самое примечательное!
    */
    SELECT colors INTO l_colors FROM color_models
    WHERE model_type = 'RGB';
    ...
END;
```

Удобно, верно? Обратите внимание на несколько важных моментов:

- При выборке значений из базы данных индексами коллекции `l_colors` управляет Oracle, а не программист.
- Начальное значение индекса, присваиваемое Oracle, равно 1 (а не 0, как в некоторых других языках). Далее индексы увеличиваются на 1; коллекция всегда заполняется плотно (или является пустой).
- Операция присваивания удовлетворяет требованию инициализации локальной переменной перед присваиванием значений элементам. Мы не инициализировали `l_colors` с помощью конструктора, но PL/SQL известно, как ее обрабатывать.

Измененное содержимое коллекции с такой же легкостью записывается обратно в базу данных. Давайте шутки ради создадим цветовую модель Fuschia-Green-Blue:

```
DECLARE
    color_tab color_tab_t;
BEGIN
    SELECT colors INTO color_tab FROM color_models
    WHERE model_type = 'RGB';

    FOR element IN 1..color_tab.COUNT
    LOOP
        IF color_tab(element) = 'RED'
        THEN
            color_tab(element) := 'FUSCHIA';
        END IF;
    END LOOP;

    /* Самая интересная часть примера. Достаточно всего
    || одной инструкции вставки - и вся вложенная таблица
```

```
|| отправляется обратно в базу данных, в таблицу color_models. */  
  
INSERT INTO color_models VALUES ('FGB', color_tab);  
END;
```

Интеграция массивов VARRAY

Резонно спросить, реализована ли интеграция баз данных с PL/SQL и для массивов VARRAY? Конечно, хотя у нее есть свои особенности.

Прежде всего, при записи и выборке содержимого вложенной таблицы из базы данных Oracle не гарантирует сохранения порядка следования элементов таблицы. И это вполне логично, потому что сервер при этом скрытно сохраняет вложенные данные во вспомогательной таблице, а мы все знаем, что реляционные базы данных не обращают внимания на порядок строк. А вот при чтении и записи содержимого VARRAY порядок элементов *сохраняется*.

Сохранение порядка следования элементов — очень полезная особенность массивов VARRAY. Порядок очень часто несет определенную информационную нагрузку. Если, допустим, элементы должны храниться в порядке их важности, их следует записать в столбец типа VARRAY. При выполнении каждой операции чтения вы будете получать элементы из этого столбца в той последовательности, в которой они были записаны. Если же придерживаться «чистой» реляционной модели, потребуется два столбца: для самих данных и для целочисленных значений, определяющих степень важности элементов.

Указанное свойство массивов VARRAY открывает широкие возможности для написания новых интересных функций. Например, вы можете запрограммировать вставку дополнительного элемента в начало списка со сдвигом всех остальных элементов.

Существует еще одно отличие структур VARRAY от вложенных таблиц, связанное с интеграцией базы данных и PL/SQL: некоторые инструкции SELECT для выборки вложенных таблиц нельзя использовать с массивами VARRAY «как есть» — в них необходимо внести изменения (примеры приведены в разделе «Работа с коллекциями в SQL»).

Заполнение коллекций данными

Только что инициализированная коллекция пуста. В ней нет ни одного определенного элемента. Элемент определяется при присваивании ему значения. Присваивание выполняется стандартной командой присваивания PL/SQL, выборкой данных из одной или нескольких реляционных таблиц в коллекцию, конструкцией RETURNING BULK COLLECT или агрегатным присваиванием (по сути, копированием одной коллекции в другую).

При работе с ассоциативным массивом значение (соответствующего типа) может быть присвоено по любому допустимому индексу в коллекции. Если ассоциативный массив индексируется целыми числами, то значения индекса должны лежать в интервале от -2^{31} и $2^{31}-1$. Простая операция присваивания создает элемент и размещает в нем значение.

В отличие от ассоциативных массивов, вложенные таблицы и VARRAY не позволяют присваивать значения по произвольным индексам. Их индексы (по крайней мере в исходном состоянии) представляют собой монотонно возрастающие целые числа, присваиваемые ядром PL/SQL. Таким образом, при инициализации n элементов им присваиваются индексы от 1 до n — и только этим элементам могут быть присвоены значения.

Прежде чем пытаться присваивать значение по индексу вложенной таблицы или VARRAY, необходимо убедиться в том, что (1) коллекция была инициализирована, и (2) индексы были определены. Для создания новых индексов во вложенных таблицах и VARRAY используется оператор EXTEND (см. ранее в этой главе).

Использование команды присваивания

Значения элементов коллекции могут задаваться стандартной командой присваивания PL/SQL:

```
countdown_test_list (43) := 'Internal pressure';
company_names_table (last_name_row + 10) := 'Johnstone Clingers';
```

Тот же синтаксис применяется для присваивания элементу коллекции целой записи или составного типа данных:

```
DECLARE
  TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
  l_emps emp_copy_t := emp_copy_t();
  l_emprec employees%ROWTYPE;
BEGIN
  l_emprec.last_name := 'Steven';
  l_emprec.salary := 10000;
  l_emps.EXTEND;
  l_emps (l_emps.LAST) := l_emprec;
END;
```

Если структура данных в правой стороне оператора присваивания соответствует типу коллекции, присваивание будет выполнено без ошибок.

Какие значения индексов можно использовать?

При присваивании данных ассоциативному массиву необходимо задать позицию (значение индекса) в коллекции. Тип значения и диапазон значений, используемых для обозначения этой позиции, зависят от способа определения секции INDEX BY ассоциативного массива. Они перечислены в следующей таблице.

| Секция INDEX BY | Минимальное значение | Максимальное значение |
|-------------------------|--|--|
| INDEX BY BINARY_INTEGER | -2^{31} | $2^{31}-1$ |
| INDEX BY PLS_INTEGER | -2^{31} | $2^{31}-1$ |
| INDEX BY SIMPLE_INTEGER | -2^{31} | $2^{31}-1$ |
| INDEX BY NATURAL | 0 | $2^{31}-1$ |
| INDEX BY POSITIVE | 1 | $2^{31}-1$ |
| INDEX BY SIGNTYPE | -1 | 1 |
| INDEX BY VARCHAR2(n) | Любая строка в пределах заданной длины | Любая строка в пределах заданной длины |

Также можно выполнить индексирование по любому из субтипов или воспользоваться типом, привязанным к столбцу базы данных VARCHAR2 (например, *имя_таблицы.имя_столбца*%TYPE).

Агрегатное присваивание

Также возможно «агрегатное присваивание» содержимого всей коллекции другой коллекции точно такого же типа. Пример:

```
1 DECLARE
2   TYPE name_t IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
3   old_names name_t;
4   new_names name_t;
5 BEGIN
6   /* Присваивание значений элементам таблицы old_names */
7   old_names(1) := 'Smith';
8   old_names(2) := 'Harrison';
9
10  /* Присваивание значений элементам таблицы new_names */
```

```

11     new_names(111) := 'Hanrahan';
12     new_names(342) := 'Blimey';
13
14     /* Перенос значений из старой коллекции в новую */
15     old_names := new_names;
16
17     /* Выводит 'Hanrahan' */
18     DBMS_OUTPUT.PUT_LINE (
19         old_names.FIRST || ': ' || old_names(old_names.FIRST));
20 END;
```

Результат:

111: Hanrahan

Присваивание уровня коллекции полностью заменяет ранее определенные элементы. В нашем примере перед последним, агрегатным присваиванием в коллекции `old_names` определены только строки 1 и 2. После присваивания данные будут содержать только элементы 111 и 342.

Присваивание из реляционной таблицы

Значения элементов коллекции также могут присваиваться на основании выборки данных из реляционной таблицы. Следующий пример демонстрирует различные способы копирования данных из реляционных таблиц в коллекции. Неявная конструкция `SELECT INTO` применяется для выборки одной строки данных в коллекцию:

```

DECLARE
    TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
    l_emps emp_copy_t := emp_copy_t();
BEGIN
    l_emps.EXTEND;
    SELECT *
        INTO l_emps (1)
        FROM employees
        WHERE employee_id = 7521;
END;
```

Для копирования нескольких строк в коллекцию с последовательным заполнением элементов можно воспользоваться курсорным циклом `FOR`:

```

DECLARE
    TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
    l_emps emp_copy_t := emp_copy_t();
BEGIN
    FOR emp_rec IN (SELECT * FROM employees)
    LOOP
        l_emps.EXTEND;
        l_emps(l_emps.LAST) := emp_rec;
    END LOOP;
END;
```

Также возможно использовать цикл `FOR` с курсором для перемещения нескольких строк в коллекцию с *непоследовательным* заполнением. В этом случае я переключаюсь на использование ассоциативного массива и поэтому могу выполнять произвольное присваивание — то есть с указанием значения первичного ключа каждой строки базы данных как номера строки моей коллекции:

```

DECLARE
    TYPE emp_copy_t IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
    l_emps emp_copy_t;
BEGIN
    FOR emp_rec IN (SELECT * FROM employees)
    LOOP
```

продолжение ➤

```

        l_emps (emp_rec.employee_id) := emp_rec;
    END LOOP;
END;
```

А еще можно воспользоваться конструкцией **BULK COLLECT** (см. главу 21) для выборки всех строк таблицы за одну операцию присваивания, с сохранением данных в любом из трех типов коллекций. При использовании вложенной таблицы или **VARRAY** явная инициализация коллекции *не нужна*. Пример:

```

DECLARE
    TYPE emp_copy_nt IS TABLE OF employees%ROWTYPE;
    l_emps emp_copy_nt;
BEGIN
    SELECT * BULK COLLECT INTO l_emps FROM employees;
END;
```

Преимущества непоследовательного заполнения коллекций

Если вам доводилось работать с традиционными массивами, идея непоследовательного заполнения массива может показаться странной. Казалось бы, для чего это может понадобиться?

Во многих приложениях вам приходится снова и снова писать и выполнять одинаковые вопросы. В некоторых случаях запросы осуществляют выборку статических данных (скажем, редко изменяющихся кодов и описаний). Но если данные остаются неизменными — особенно во время одного пользовательского сеанса, — зачем многократно извлекать информацию из базы данных? Даже если данные кэшируются в области SGA (System Global Area), вам все равно придется обратиться к SGA, убедиться в том, что запрос уже разобран, найти эту информацию в буферах данных и вернуть ее в область сеанса (PGA).

Предлагается следующее решение проблемы: для заданной статической таблицы пользователь должен запрашивать любую конкретную строку не более одного раза за сеанс. После первого раза строка сохраняется в PGA для будущих запросов. Такая конфигурация очень легко реализуется с использованием коллекций.

Рассмотрим пример: в таблице **hairstyles** хранятся числовые коды (первичный ключ) и описания стрижек. Эти данные не устаревают и изменяются относительно редко.

Ниже представлено тело пакета, использующего коллекцию для кэширования пар «код-описание» и тем самым сокращающего количество обращений к базе:

```

1 PACKAGE BODY justonce
2 IS
3     TYPE desc_t
4     IS
5         TABLE OF hairstyles.description%TYPE
6         INDEX BY PLS_INTEGER;
7
8     descriptions  desc_t;
9
10    FUNCTION description (code_in IN hairstyles.code%TYPE)
11    RETURN hairstyles.description%TYPE
12    IS
13        return_value  hairstyles.description%TYPE;
14
15        FUNCTION desc_from_database
16        RETURN hairstyles.description%TYPE
17        IS
18            l_description  hairstyles.description%TYPE;
19        BEGIN
20            SELECT description
```



```

21         INTO l_description
22         FROM hairstyles
23         WHERE code = code_in;
24         RETURN l_description;
25     END;
26 BEGIN
27     RETURN descriptions (code_in);
28 EXCEPTION
29     WHEN NO_DATA_FOUND
30     THEN
31         descriptions (code_in) := desc_from_database ();
32     RETURN descriptions (code_in);
33 END;
34 END justonce;

```

Следующая таблица описывает интересные аспекты программы.

| Строки | Описание |
|--------|--|
| 3–8 | Объявление типа и экземпляра коллекции для хранения кэшированных описаний |
| 10–11 | Заголовок функции выборки. В нем нет абсолютно ничего интересного; ничто не указывает на то, что функция делает хоть что-то помимо типичного запроса к базе данных для получения описания по числовому коду. Реализация скрыта, как мы и хотели |
| 15–25 | Вполне традиционный запрос к базе данных. Но в данном случае он реализуется приватной функцией внутри главной функции, что вполне уместно, потому что в этом примере нас интересует другое |
| 27 | Весь исполняемый раздел! Просто возвращаем описание, хранящееся в строке, определяемой числовым кодом. При первом выполнении этой функции для заданного кода строка не определена, поэтому PL/SQL инициирует исключение NO_DATA_FOUND (строки 28–31). Но для всех последующих обращений с данным кодом элемент определен, а функция немедленно возвращает запрашиваемые данные |
| 29–32 | Данные еще не запрашивались в ходе текущего сеанса. Перехватываем ошибку, получаем описание из базы данных и помещаем его в коллекцию, после чего возвращаем значение |

Итак, к каким последствиям приводит кэширование? Тесты на моем компьютере показали, что выполнение 10 000 запросов к таблице `hairstyles` заняло менее двух секунд. Несомненно, это хорошая скорость. Однако для получения той же информации 10 000 раз с использованием приведенной функции потребовалось около 0,1 секунды. Улучшение больше чем на порядок!

Напоследок несколько замечаний по кэшированию в коллекциях:

- Это решение является классическим компромиссом между затратами вычислительных ресурсов и памяти. Каждый сеанс использует собственную копию коллекции (данные программы хранятся в PGA). При 10 000 пользователей общие затраты памяти на 10 000 небольших кэшей будут довольно значительными.
- Кэширование часто оказывается уместным в следующих сценариях: небольшие статические таблицы в многопользовательских приложениях; большие статические таблицы, в которых конкретному пользователю доступна лишь небольшая часть данных; манипуляции с большими таблицами в пакетных процессах (простая команда `CONNECT`, которая может занимать много памяти).

Концепции и варианты реализации кэширования более подробно рассматриваются в главе 21.

Обращение к данным в коллекциях

Вероятно, размещение информации в коллекции имеет смысл только в том случае, если вы собираетесь использовать эти данные или обращаться к ним. При обращениях к данным в коллекциях следует учитывать несколько обстоятельств:

- При попытке чтения по неопределенному индексу в коллекции база данных инициирует исключение `NO_DATA_FOUND`. Из этого следует, что вам стоит по возможности избегать использования циклов `FOR` со счетчиком для перебора коллекции, если только вы не уверены, что она плотно заполнена — и всегда будет оставаться таковой (то есть между `FIRST` и `LAST` нет неопределенных значений индексов). Если коллекция заполнена неплотно, при первом же обращении к «дыре» между значениями, возвращаемыми методами `FIRST` и `LAST`, произойдет сбой с исключением `NO_DATA_FOUND`.
- При попытке чтения строки, выходящей за пределы расширенных (посредством `EXTEND`) строк таблицы или `VARRAY`, база данных инициирует следующее исключение: *ORA-06533: Subscript beyond count*.

При работе с вложенными таблицами и `VARRAY` следует постоянно следить за расширением коллекции, чтобы в ней помещались строки, которые вы хотите прочитать или присвоить.

- При попытке чтения строки, индекс которой выходит за границы определения типа `VARRAY`, база данных выдает следующее исключение: *ORA-06532: Subscript outside of limit*.

Помните, что вы всегда можете вызвать метод `LIMIT` для определения максимального допустимого количества строк в `VARRAY`. Так как индексы в таких коллекциях всегда начинаются с 1, вы можете легко определить, осталось ли в структуре данных место для дополнительной информации.

В остальном обращения к отдельным строкам коллекции не вызывают никаких сложностей: просто укажите индекс (или индексы — синтаксис коллекций с элементами-коллекциями рассматривается в разделе «Коллекции составных типов данных») после имени коллекции.

Коллекции со строковыми индексами

В Oracle9i Release 2 состав типов, которые могут использоваться для индексирования ассоциативных массивов, был значительно расширен. Тип `VARCHAR2` обладает наибольшей гибкостью и потенциальными возможностями. Поскольку с этим типом возможно индексирование по строкам, фактически появляется возможность индексирования по любым данным, которые могут быть преобразованы в строку длиной не более 32 767 байт. Следующий блок кода демонстрирует основные принципы строкового индексирования:

```
/* Файл в Сети: string_indexed.sql */
DECLARE
    SUBTYPE location_t IS VARCHAR2(64);
    TYPE population_type IS TABLE OF NUMBER INDEX BY location_t;

    l_country_population population_type;
    l_continent_population population_type;

    l_count PLS_INTEGER;
    l_location location_t;
BEGIN
    l_country_population('Greenland') := 100000;
    l_country_population('Iceland') := 750000;

    l_continent_population('Australia') := 30000000;
    l_continent_population('Antarctica') := 1000;
    l_continent_population('antarctica') := 1001;

    l_count := l_country_population.COUNT;
    DBMS_OUTPUT.PUT_LINE ('COUNT = ' || l_count);
```

```

l_location := l_continent_population.FIRST;
DBMS_OUTPUT.PUT_LINE ('FIRST row = ' || l_location);
DBMS_OUTPUT.PUT_LINE ('FIRST value = ' || l_continent_population(l_location));

l_location := l_continent_population.LAST;
DBMS_OUTPUT.PUT_LINE ('LAST row = ' || l_location);
DBMS_OUTPUT.PUT_LINE ('LAST value = ' || l_continent_population(l_location));
END;
```

Результат выполнения сценария:

```

COUNT = 2
FIRST row = Antarctica
FIRST value = 1000
LAST row = antarctica
LAST value = 1001
```

В этом коде следует обратить внимание на некоторые моменты:

- В коллекции со строковым индексированием значения, возвращаемые методами FIRST, LAST, PRIOR и NEXT, представляют собой строки, а не целые числа.
- Обратите внимание: «antarctica» стоит на последнем месте, после «Antarctica» и «Australia». Дело в том, что буквы нижнего регистра имеют большие значения ASCII-кодов, чем буквы верхнего регистра. Порядок сохранения строк в ассоциативном массиве определяется набором символов.
- Между коллекциями со строковым и целочисленным индексированием нет различий в синтаксисе использования.
- Я определяю подтип `location_t`, который будет использоваться как тип индексирования в моем объявлении типа коллекции, а также при объявлении переменной `l_location`. При работе с коллекциями, индексруемыми строками (и особенно многоуровневыми коллекциями), подтипы напоминают, какие данные используются в качестве значений индексов.

Далее приводятся другие примеры, демонстрирующие применение этой возможности.

Упрощение логики алгоритмов при использовании строковых индексов

Разумное использование коллекций со строковым индексированием может сильно упростить ваши программы; по сути сложность выносится из алгоритмов в структуру данных, которые и выполняют всю «черную работу» (вернее, ее выполняет база данных). Следующий пример наглядно демонстрирует это перемещение сложности.

В 2006 и 2007 годах я руководил проектом по созданию программы автоматизации тестирования PL/SQL — Quest Code Tester — для Oracle. Одно из важнейших преимуществ этой программы заключалось в том, что она генерировала тестовый пакет по описаниям ожидаемого поведения программы. В процессе генерирования тестового кода необходимо отслеживать имена объявленных переменных, чтобы случайно не объявить другую переменную с тем же именем.

Первая версия пакета `string_tracker` выглядела примерно так:

```

/* Файл в Сети: string_tracker0.pkg */
1 PACKAGE BODY string_tracker
2 IS
3     SUBTYPE name_t IS VARCHAR2 (32767);
4     TYPE used_aat IS TABLE OF name_t INDEX BY PLS_INTEGER;
5     g_names_used used_aat;
6
7     PROCEDURE mark_as_used (variable_name_in IN name_t) IS
8     BEGIN
9         g_names_used (g_names_used.COUNT + 1) := variable_name_in;
```

продолжение ➤

```

10     END mark_as_used;
11
12     FUNCTION string_in_use (variable_name_in IN name_t) RETURN BOOLEAN
13     IS
14         c_count    CONSTANT PLS_INTEGER := g_names_used.COUNT;
15         l_index     PLS_INTEGER := g_names_used.FIRST;
16         l_found     BOOLEAN      := FALSE;
17     BEGIN
18         WHILE (NOT l_found AND l_index <= c_count)
19         LOOP
20             l_found := variable_name_in = g_names_used (l_index);
21             l_index := l_index + 1;
22         END LOOP;
23
24         RETURN l_found;
25     END string_in_use;
26 END string_tracker;

```

В таблице поясняются наиболее интересные места в теле пакета.

| Строки | Описание |
|--------|--|
| 3–5 | Объявление коллекции строк, индексируемой целыми числами, для хранения списка уже использованных имен переменных |
| 7–10 | Имя переменной добавляется в конец массива; тем самым оно помечается как «используемое» |
| 12–25 | Перебор коллекции в поисках совпадения имени. Если имя обнаруживается, то перебор завершается с возвращением TRUE. В противном случае возвращается значение FALSE (строка не используется) |

Конечно, это далеко не самый большой и сложный пакет. Тем не менее объем кода больше необходимого, а его выполнение занимает больше процессорного времени, чем необходимо. Как упростить код и ускорить его? Нужно воспользоваться коллекцией со строковым индексированием.

Вторая версия пакета string_tracker:

```

/* Файл в Сети: string_tracker1.pkg */
1  PACKAGE BODY string_tracker
2  IS
3      SUBTYPE name_t IS VARCHAR2 (32767);
4      TYPE used_aat IS TABLE OF BOOLEAN INDEX BY name_t;
5      g_names_used used_aat;
6
7      PROCEDURE mark_as_used (variable_name_in IN name_t) IS
8      BEGIN
9          g_names_used (variable_name_in) := TRUE;
10     END mark_as_used;
11
12     FUNCTION string_in_use (variable_name_in IN name_t) RETURN BOOLEAN
13     IS
14     BEGIN
15         RETURN g_names_used.EXISTS (variable_name_in);
16     END string_in_use;
17 END string_tracker;

```

| Строки | Описание |
|--------|---|
| 3–5 | На этот раз я объявляю коллекцию логических значений, индексируемых строками. Вообще говоря, неважно, какие именно данные содержатся в коллекции — я могу создать коллекцию логических флагов, дат, чисел, документов XML и т. д. Как вы вскоре увидите, важно лишь значение индекса |
| 7–10 | Я снова помечаю строку как используемую, но в этой версии имя переменной служит значением индекса, а не присоединяется в конец коллекции. Я связываю с индексом значение TRUE, но как было сказано выше, это значение может быть любым: NULL, TRUE, FALSE. Это несущественно, потому что... |

| Строки | Описание |
|--------|--|
| 12–16 | Чтобы определить, было ли использовано имя переменной ранее, достаточно вызвать метод EXISTS для имени переменной. Если для данного значения индекса определен элемент, значит, имя уже использовано. Другими словами, я вообще не обращаюсь к значению, связанному с этим индексом (и оно вообще ни на что не влияет) |

Не правда ли, решение выглядит просто и элегантно? Мне не нужно писать код перебора коллекции в поисках совпадений. Вместо этого я сразу указываю значение индекса и сразу получаю ответ.

Из работы над пакетом `string_tracker` я вынес важный урок: если во время разработки мне приходится писать алгоритмы поиска в коллекции, перебирающие элемент за элементом, стоит изменить эту коллекцию (или создать другую) и перейти на строковое индексирование, предотвращающее «полный перебор». Программа получается более компактной и эффективной, а ее сопровождение в будущем упрощается.

Моделирование первичных ключей и уникальных индексов

Среди интересных применений строкового индексирования стоит выделить моделирование первичных ключей и уникальных индексов реляционных таблиц в коллекциях. Допустим, мне требуется выполнить некие серьезные вычисления для обработки информации о работниках в моей программе. Для этого нужно работать с данными группы работников и часто проводить поиск по коду работника, фамилии и адресу электронной почты.

Вместо того чтобы запрашивать данные из базы, я могу кэшировать их в коллекциях, а затем работать с этими данными с существенно большей эффективностью. Возможная реализация могла бы выглядеть так:

```
DECLARE
  c_delimiter CONSTANT CHAR (1) := '^';
  TYPE strings_t IS TABLE OF employees%ROWTYPE
      INDEX BY employees.email%TYPE;
  TYPE ids_t IS TABLE OF employees%ROWTYPE
      INDEX BY PLS_INTEGER;
  by_name      strings_t;
  by_email     strings_t;
  by_id        ids_t;
  ceo_name     employees.last_name%TYPE
      := 'ELLISON' || c_delimiter || 'LARRY';
  PROCEDURE load_arrays
  IS
  BEGIN
    /* Заполнение всех трех массивов по строкам таблицы. */
    FOR rec IN (SELECT *
                FROM employees)
    LOOP
      by_name (rec.last_name || c_delimiter || rec.first_name) := rec;
      by_email (rec.email) := rec;
      by_id (rec.employee_id) := rec;
    END LOOP;
  END;
BEGIN
  load_arrays;

  /* Выборка информации по имени или идентификатору. */
  IF by_name (ceo_name).salary > by_id (7645).salary
  THEN
    make_adjustment (ceo_name);
  END IF;
END;
```

Производительность коллекций со строковым индексированием

Какую цену приходится платить за использование строкового индексирования вместо целочисленного? Все зависит исключительно от длины строк. При использовании строкового индексирования база данных берет строку и «хеширует» (преобразует) ее в целочисленное значение. Таким образом, дополнительные затраты определяются быстроедействием хеширующей функции, а также необходимостью разрешения возможных конфликтов, так как уникальность результатов хеширования не гарантируется — известно лишь то, что ее вероятность *чрезвычайно высока*.

На моем компьютере тестирование дало следующий результат (см. сценарий `assoc_array_perf.tst` на сайте книги):

Сравнение строкового и целочисленного индексирования. Итерации = 10000 Длина = 100
Индексирование по PLS_INTEGER Время: 4.26 с.
Индексирование по VARCHAR2 Время: 4.75 с.
Сравнение строкового и целочисленного индексирования. Итерации = 10000 Длина = 1000
Индексирование по PLS_INTEGER Время: 4.24 с.
Индексирование по VARCHAR2 Время: 6.4 с.
Сравнение строкового и целочисленного индексирования. Итерации = 10000 Длина = 10000
Индексирование по PLS_INTEGER Время: 4.06 с.
Индексирование по VARCHAR2 Время: 24.63 с.

Вывод: при относительно небольших строках (100 символов и менее) нет существенных различий по производительности между строковым и целочисленным индексированием. Однако с увеличением длины строк затраты на хеширование существенно возрастают. Будьте внимательны при выборе строк, используемых в качестве индексов!

Другие примеры коллекций со строковым индексированием

Как вы видели в примере выборки информации о работниках, построение эффективных точек входа для кэшированных данных, загруженных из реляционной таблицы, не требует большого объема кода. Тем не менее чтобы дополнительно упростить реализацию этого приема в ваших приложениях, я написал программу, которая генерирует такой код за вас.

Файл `gena.sp` на сайте книги получает имя таблицы в аргументе. На основании информации, хранящейся в словаре данных этой таблицы (первичный ключ и уникальные индексы), он генерирует реализацию кэширования для этой таблицы. Затем он заполняет коллекцию на основании целочисленного первичного ключа и дополнительную коллекцию для каждого уникального индекса, определенного для таблицы (с индексированием по PLS_INTEGER или VARCHAR2 в зависимости от типа(-ов) столбца(-ов) в индексе).

Кроме того, файл `summer_reading.pkg`, также доступный на сайте книги, содержит пример использования ассоциативных массивов, индексируемых по VARCHAR2, для работы со списками в программах PL/SQL.

Обратите внимание: «antarctica» стоит на последнем месте, после «Antarctica» и «Australia». Дело в том, что буквы нижнего регистра имеют большие значения ASCII-кодов, чем буквы верхнего регистра. Порядок сохранения строк в ассоциативном массиве определяется набором символов.

Между коллекциями со строковым и целочисленным индексированием нет различий в синтаксисе использования.

Коллекции составных типов данных

В Oracle9i Release 2 появилась возможность определения типов коллекций произвольной сложности. Поддерживаются следующие структуры:

- **Коллекции записей, определяемых на базе таблиц с атрибутом %ROWTYPE.** Эти структуры позволяют легко и быстро моделировать реляционные таблицы в программах PL/SQL.
- **Коллекции пользовательских записей.** Поля записи также могут относиться к скалярным или составным типам данных. Например, можно определить коллекцию записей, одно из полей которых само по себе является коллекцией.
- **Коллекции объектных типов.** Элементы коллекции могут относиться к любому из объектных типов (разновидность объектно-ориентированных классов, используемая в Oracle, — см. главу 26), ранее определенных инструкцией CREATE TYPE. Так же легко определяются коллекции LOB, документов XML и т. д.
- **Коллекции коллекций.** Возможно определение многоуровневых коллекций, в том числе коллекций коллекций и коллекций типов данных, содержащих другие коллекции в своих атрибутах или полях.

Рассмотрим пример использования каждой из этих разновидностей.

Коллекции записей

Коллекция записей определяется с указанием типа записи (либо посредством %ROWTYPE, либо для типа записи, определяемого пользователем) в секции TABLE OF определения коллекции. Эта методика применяется только к типам коллекций, объявляемым в программе PL/SQL. Вложенные таблицы и типы VARRAY, определяемые в базе данных, не могут обращаться к структурам записей %ROWTYPE.

Пример коллекции записей, созданных на базе пользовательского типа записи:

```
PACKAGE compensation_pkg
IS
    TYPE reward_rt IS RECORD (
        nm VARCHAR2(2000), sal NUMBER, comm NUMBER);
    TYPE reward_tt IS TABLE OF reward_rt INDEX BY PLS_INTEGER;
END compensation_pkg;
```

С такими определениями типов в спецификации пакета я могу объявлять коллекции в других программах:

```
DECLARE
    holiday_bonuses compensation_pkg.reward_tt;
```

Коллекции записей особенно удобны для создания в памяти коллекций, имеющих такую же структуру (и по крайней мере частично — содержащих те же данные), что и таблицы базы данных. Для чего это может понадобиться? Допустим, каждое воскресенье в 3 часа ночи я запускаю пакетный процесс для таблиц, измененных за последнюю неделю. Мне нужно провести основательный анализ с несколькими проходами по данным таблицы. Конечно, данные можно многократно загружать из базы данных, но это относительно медленный процесс, создающий значительную вычислительную нагрузку.

Также можно скопировать данные из таблицы (или таблиц) в коллекцию, чтобы иметь возможность быстрее (и более произвольно) перемещаться по итоговому набору. Фактически я моделирую двусторонние курсоры в своем коде PL/SQL.

Если вы захотите скопировать данные в коллекции и работать с ними в программе, существует два основных подхода к реализации этой логики:

- Встроить весь код коллекций в основную программу.
- Создать отдельный пакет для инкапсуляции доступа к данным в коллекции.

В большинстве случаев я выбираю второй подход. Другими словами, я считаю полезным создавать отдельные, четко определенные API с высоким потенциалом повторного

использования для сложных структур и логики. Ниже приведена спецификация пакета для моей модели двустороннего курсора:

```
/* Файл в Сети: bidir.pkg */
PACKAGE bidir
IS
    FUNCTION rowforid (id_in IN employees.employee_id%TYPE)
        RETURN employees%ROWTYPE;
    FUNCTION firstrow RETURN PLS_INTEGER;
    FUNCTION lastrow RETURN PLS_INTEGER;
    FUNCTION rowCount RETURN PLS_INTEGER;
    FUNCTION end_of_data RETURN BOOLEAN;
    PROCEDURE setrow (nth IN PLS_INTEGER);
    FUNCTION currrow RETURN employees%ROWTYPE;
    PROCEDURE nextrow;
    PROCEDURE prevrow;
END;
```

Как использовать этот API? Ниже приведен пример программы, использующей этот API для чтения итогового набора для таблицы `employees` — сначала в прямом, а затем в обратном направлении:

```
/* Файл в Сети: bidir.tst */
DECLARE
    l_employee employees%ROWTYPE;
BEGIN
    LOOP
        EXIT WHEN bidir.end_of_data;
        l_employee := bidir.currrow;
        DBMS_OUTPUT.put_line (l_employee.last_name);
        bidir.nextrow;
    END LOOP;

    bidir.setrow (bidir.lastrow);
    LOOP
        EXIT WHEN bidir.end_of_data;
        l_employee := bidir.currrow;
        DBMS_OUTPUT.put_line (l_employee.last_name);
        bidir.prevrow;
    END LOOP;
END;
```

Наблюдательный читатель спросит: когда коллекция заполняется данными? Или еще лучше: где сама коллекция? В представленном коде нет никаких признаков коллекции.

Начнем со второго вопроса. Коллекция не видна потому, что я скрыл ее в спецификации пакета. Пользователь пакета никогда не соприкасается с коллекцией и ничего не знает о ней. Собственно, для этого и создавался API. Вы просто вызываете ту или иную программу, которая выполняет за вас всю работу по перебору коллекции (набора данных).

Когда и как заполняется коллекция? На первый взгляд все кажется каким-то волшебством... пока вы не познакомитесь с пакетами в главе 18. Заглянув в тело пакета, вы найдете в нем раздел инициализации, которая выглядит так:

```
BEGIN -- инициализация пакета
    FOR rec IN (SELECT * FROM employees)
    LOOP
        g_employees (rec.employee_id) := rec;
    END LOOP;
    g_currrow := firstrow;
END;
```

Таким образом, при первом обращении к любому элементу из спецификации пакета автоматически выполняется этот код, который передает содержимое таблицы `employees`

в коллекцию `g_employees`. Когда это происходит в приведенном примере? В цикле, когда я вызываю функцию `bidir.end_of_data`, чтобы проверить, не завершился ли просмотр набора данных!



Переменная `g_singrow` определяется в теле пакета и поэтому не указывается в приведенной выше спецификации.

Я рекомендую просмотреть реализацию пакета. Ее код прост и понятен, а в некоторых ситуациях такое решение способно принести огромную пользу.

Коллекции объектов и других составных типов

В качестве типа данных команды `TYPE` коллекции может использоваться объектный тип, `LOB`, документ XML и вообще практически любой действительный тип PL/SQL. Синтаксис определения этих коллекций не отличается, но способ выполнения операций с их содержимым может быть достаточно сложным (в зависимости от используемого типа). За дополнительной информацией об объектных типах Oracle обращайтесь к главе 26.

Пример работы с коллекциями объектов:

```
/* Файл в Сети: object_collection.sql */
CREATE TYPE pet_t IS OBJECT (
  tag_no    INTEGER,
  name      VARCHAR2 (60),
  MEMBER FUNCTION set_tag_no (new_tag_no IN INTEGER) RETURN pet_t);
/
DECLARE
  TYPE pets_t IS TABLE OF pet_t;
  pets      pets_t :=
    pets_t (pet_t (1050, 'Sammy'), pet_t (1075, 'Mercury'));
BEGIN
  FOR indx IN pets.FIRST .. pets.LAST
  LOOP
    DBMS_OUTPUT.put_line (pets (indx).name);
  END LOOP;
END;
/
```

Результат:

```
Sammy
Mercury
```

После того как объектный тип будет определен, я могу объявить на основании определения новую коллекцию, а затем заполнить ее экземплярами объектного типа. С такой же легкостью можно объявлять коллекции `LOB`, `XMLType` и т. д. Все обычные правила, действующие для переменных этих типов данных, также распространяются и на отдельные строки коллекций этого типа.

Многоуровневые коллекции

В Oracle9i Database Release 2 появилась возможность создания *вложенных* коллекций, которые также называются *многоуровневыми*. Начнем с примера, а потом обсудим использование данной возможности в ваших приложениях.

Предположим, я хочу построить систему для управления информацией о домашних животных. Кроме стандартной информации (порода, имя и т. д.), в системе должны храниться сведения о посещениях ветеринара. Для хранения этой информации создается объектный тип:

```
CREATE TYPE vet_visit_t IS OBJECT (
    visit_date DATE,
    reason      VARCHAR2 (100)
);
/
```

Обратите внимание: объекты, создаваемые на основе этого типа, не связываются с объектом домашнего животного (то есть внешним ключом таблицы домашних животных или объектом). Вскоре вы увидите, почему это не нужно. На следующем шаге создается вложенная таблица посещений (предполагается, что посещения происходят не реже одного раза в год):

```
CREATE TYPE vet_visits_t IS TABLE OF vet_visit_t;
/
```

С такими определениями структур данных мы можем создать объектный тип для хранения информации о домашних животных:

```
CREATE TYPE pet_t IS OBJECT (
    tag_no    INTEGER,
    name      VARCHAR2 (60),
    petcare   vet_visits_t,
    MEMBER FUNCTION set_tag_no (new_tag_no IN INTEGER) RETURN pet_t);
/
```

Объектный тип содержит три атрибута и один метод. С любым объектом, созданным на базе этого типа, связывается регистрационный номер, имя и список посещений ветеринара. Для изменения регистрационного номера животного вызывается программа `set_tag_no`.

Итак, я объявил объектный тип, содержащий атрибут вложенной таблицы. Для хранения информации о посещениях ветеринара не нужна отдельная таблица базы данных; она является частью моего объекта.

А теперь мы воспользуемся возможностями многоуровневых коллекций:

```
/* Файл в Сети: multilevel_collections.sql */
1 DECLARE
2   TYPE bunch_of_pets_t
3     IS
4       TABLE OF pet_t INDEX BY PLS_INTEGER;
5
6   my_pets   bunch_of_pets_t;
7 BEGIN
8   my_pets (1) :=
9     pet_t (
10       100
11       , 'Mercury'
12       , vet_visits_t (vet_visit_t ('01-Jan-2001', 'Clip wings')
13                           , vet_visit_t ('01-Apr-2002', 'Check cholesterol')
14       )
15   );
16   DBMS_OUTPUT.put_line (my_pets (1).name);
17   DBMS_OUTPUT.put_line
18     (my_pets(1).petcare.LAST).reason);
19   DBMS_OUTPUT.put_line (my_pets.COUNT);
20   DBMS_OUTPUT.put_line (my_pets (1).petcare.LAST);
21 END;
```

Результат выполнения этой программы:

```
Mercury
Check cholesterol
1
2
```

Следующая таблица поясняет, что происходит в этом коде.

| Строки | Описание |
|--------|---|
| 2–6 | Объявление локального ассоциативного массива TYPE, в котором каждая строка содержит один объект домашнего животного. Затем я объявляю коллекцию для хранения информации о всем «зверинце» |
| 8–15 | Присваивание объекта типа pet_t по индексу 1 ассоциативного массива. Как видите, синтаксис, необходимый для работы с составными вложенными объектами такого рода, выглядит довольно устрашающе. Давайте разберемся поподробнее: для создания экземпляра типа pet_t необходимо предоставить регистрационный номер, имя и список посещений ветеринара, который представляет собой вложенную таблицу. Чтобы создать вложенную таблицу типа vet_visits_t, я должен вызвать ассоциированный конструктор (с тем же именем). При этом либо указывается пустой список, либо вложенная таблица инициализируется некими значениями (строки 8–9). Каждая строка коллекции vet_visits_t представляет собой объект типа vet_visit_t, поэтому я снова должен использовать конструктор объекта и передать значение для каждого атрибута (дата и причина посещения) |
| 16 | Вывод значения атрибута name объекта домашнего животного из строки 1 ассоциативного массива my_pets |
| 17–18 | Вывод значения атрибута reason объекта посещения ветеринара из строки 2 вложенной таблицы, которая, в свою очередь, хранится по индексу 1 ассоциативного массива my_pets. Как видите, и описание, и код получаются довольно громоздкими |
| 19–21 | Демонстрация использования методов коллекций (в данном случае COUNT и LAST) для внешних и вложенных коллекций |

В этом примере нам повезло работать с коллекциями, которые на каждом уровне используют имена: ассоциативным массивом my_pets и вложенной таблицей petcare. Как показывает следующий пример, так бывает не всегда.

**Безымянные многоуровневые коллекции:
моделирование многомерных массивов**

Вложенные многоуровневые коллекции могут использоваться для моделирования многомерных массивов в PL/SQL. Многомерные коллекции объявляются за несколько шагов, при этом на каждом шаге добавляется новое измерение (что сильно отличается от синтаксиса объявления массивов в 3GL).

Мы начнем с простого примера, а затем разберем реализацию обобщенного пакета трехмерных массивов. Предположим, я хочу хранить данные температуры в некотором трехмерном пространстве, упорядоченном с использованием системы координат (X, Y, Z). Следующий блок демонстрирует необходимую последовательность объявлений:

```
DECLARE
  SUBTYPE temperature IS NUMBER;
  SUBTYPE coordinate_axis IS PLS_INTEGER;
  TYPE temperature_x IS TABLE OF temperature INDEX BY coordinate_axis;
  TYPE temperature_xy IS TABLE OF temperature_x INDEX BY coordinate_axis;
  TYPE temperature_xyz IS TABLE OF temperature_xy INDEX BY coordinate_axis;
  temperature_3d temperature_xyz;
BEGIN
  temperature_3d (1) (2) (3) := 45;
END;
```

Имена типов и подтипов поясняют смысл содержимого основной коллекции (temperature_3d), типов коллекций (temperature_x, temperature_xy, temperature_xyz) и индексов (coordinate_axis).

Хотя выбор имен четко указывает, какие данные содержатся в каждом из типов коллекций и для чего они предназначены, при обращении к элементам по индексу такой же ясности нет; иначе говоря, в каком порядке задаются измерения? Из кода неочевидно, присваивается ли температура 45° точке (X:1, Y:2, Z:3) или (X:3, Y:2, Z:1).

А теперь мы перейдем к более общей обработке трехмерного массива. Пакет `multdim` позволяет объявить трехмерный массив с возможностью чтения и записи отдельных ячеек. Здесь я создаю простой пакет, инкапсулирующий операции с трехмерной ассоциативной таблицей с элементами `VARCHAR2`, индексируемый по всем измерениям значением `PLS_INTEGER`. Следующие объявления содержат основные структурные элементы пакета:

```
/* Файлы в Сети: multdim.pkg, multdim.tst, multdim2.pkg */
CREATE OR REPLACE PACKAGE multdim
IS
    TYPE dim1_t IS TABLE OF VARCHAR2 (32767) INDEX BY PLS_INTEGER;
    TYPE dim2_t IS TABLE OF dim1_t INDEX BY PLS_INTEGER;
    TYPE dim3_t IS TABLE OF dim2_t INDEX BY PLS_INTEGER;
    PROCEDURE setcell (
        array_in  IN OUT  dim3_t,
        dim1_in   PLS_INTEGER,
        dim2_in   PLS_INTEGER,
        dim3_in   PLS_INTEGER,
        value_in  IN      VARCHAR2
    );
    FUNCTION getcell (
        array_in  IN  dim3_t,
        dim1_in   PLS_INTEGER,
        dim2_in   PLS_INTEGER,
        dim3_in   PLS_INTEGER
    )
    RETURN VARCHAR2;
    FUNCTION EXISTS (
        array_in  IN  dim3_t,
        dim1_in   PLS_INTEGER,
        dim2_in   PLS_INTEGER,
        dim3_in   PLS_INTEGER
    )
    RETURN BOOLEAN;
```

Я последовательно определяю три типа коллекций:

- `TYPE dim1_t` — одномерная ассоциативная таблица с элементами `VARCHAR2`.
- `TYPE dim2_t` — ассоциативная таблица с элементами `dim1_t`.
- `TYPE dim3_t` — ассоциативная таблица с элементами `dim2_t`.

Таким образом, трехмерное пространство моделируется как ячейки коллекции плоскостей, каждая из которых моделируется коллекцией линий. Такое представление соответствует здравому смыслу, что является признаком удачной модели. Конечно, мои коллекции разрежены и конечны, тогда как геометрическое трехмерное пространство считается плотным и бесконечным, так что у модели есть свои ограничения. Но для моих целей актуально только конечное подмножество точек трехмерного пространства, так что модель можно считать адекватной.

Я также определил для своего типа трехмерной коллекции простейший интерфейс чтения и записи ячеек, а также средства проверки существования значения заданной ячейки в коллекции.

Программный интерфейс `multdim`

Рассмотрим основные компоненты интерфейса. Процедура записи значения в ячейку трехмерного массива по координатам не могла бы быть проще:

```
PROCEDURE setcell (
    array_in  IN OUT  dim3_t,
    dim1_in   PLS_INTEGER,
    dim2_in   PLS_INTEGER,
    dim3_in   PLS_INTEGER,
```

```

    value_in IN VARCHAR2
)
IS
BEGIN
    array_in(dim3_in )(dim2_in )(dim1_in) := value_in;
END;
```

При всей простоте этого кода инкапсуляция присваивания приносит несомненную пользу, поскольку она освобождает меня от необходимости запоминать порядок индексов. Если работать с коллекцией `dim3_t` напрямую, трудно сразу сказать, какому индексу соответствует третья координата — первому или последнему. А то, что неочевидно из кода, рано или поздно приведет к ошибкам. Тот факт, что все индексы коллекций относятся к одному типу данных, только усложняет дело, потому что смешанные команды присваивания не породят исключений, а приведут к искажению результатов где-то в цепочке. Без тщательного тестирования такие ошибки проникнут в поставляемый код, устроят хаос в данных и повредят моей репутации.

Функция получения значения ячейки тоже тривиальна, но безусловно полезна:

```

FUNCTION getcell (
    array_in IN dim3_t,
    dim1_in PLS_INTEGER,
    dim2_in PLS_INTEGER,
    dim3_in PLS_INTEGER
)
RETURN VARCHAR2
IS
BEGIN
    RETURN array_in(dim3_in )(dim2_in )(dim1_in);
END;
```

Если `array_in` не содержит ячейки, соответствующей указанным координатам, `getcell` инициирует исключение `NO_DATA_FOUND`. Но если какие-либо из переданных координат равны `NULL`, выдается следующее, уже не столь понятное исключение `VALUE_ERROR`:

```
ORA-06502: PL/SQL: numeric or value error: NULL index table key value
```

В полноценной реализации я бы дополнил этот модуль предварительной проверкой всех параметров координат, которые должны быть отличны от `NULL`. По крайней мере сообщение об ошибке информирует, что за исключение ответственно неопределенное значение индекса. Однако было бы еще лучше, если бы база данных не использовала исключение `VALUE_ERROR` для множества разных ошибочных ситуаций.

Код функции `EXISTS` уже не столь тривиален. Функция `EXISTS` должна возвращать `TRUE`, если ячейка, определяемая координатами, содержится в коллекции, и `FALSE` в противном случае:

```

FUNCTION EXISTS (
    array_in IN dim3_t,
    dim1_in PLS_INTEGER,
    dim2_in PLS_INTEGER,
    dim3_in PLS_INTEGER
)
RETURN BOOLEAN
IS
    l_value VARCHAR2(32767);
BEGIN
    l_value := array_in(dim3_in )(dim2_in )(dim1_in);
    RETURN TRUE;
EXCEPTION
    WHEN NO_DATA_FOUND THEN RETURN FALSE;
END;
```

Функция перехватывает исключение `NO_DATA_FOUND`, инициируемое при обращении к несуществующей ячейке при присваивании, и преобразует его в соответствующее логическое значение. Это очень простой и прямолинейный способ получения нужного результата, который демонстрирует творческое применение обработки исключений для управления «условной логикой» функций. Казалось бы, в данной ситуации и нужно использовать оператор `EXISTS`, но его пришлось бы вызвать для каждого уровня вложенных коллекций. Пример сценария, использующего этот пакет:

```
/* Файл в Сети: multdim.tst */
DECLARE
  my_3d_array  multdim.dim3_t;
BEGIN
  multdim.setcell (my_3d_array, 1, 5, 800, 'def');
  multdim.setcell (my_3d_array, 1, 15, 800, 'def');
  multdim.setcell (my_3d_array, 5, 5, 800, 'def');
  multdim.setcell (my_3d_array, 5, 5, 805, 'def');

  DBMS_OUTPUT.PUT_LINE (multdim.getcell (my_3d_array, 1, 5, 800));
/*
Oracle 11g Release позволяет вызывать PUT_LINE с логическими данными!
*/
  DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 1, 5, 800));
  DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 6000, 5, 800));
  DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 6000, 5, 807));

/*
Если вы не работаете с Oracle 11g Release 2, используйте процедуру,
созданную в bpl.sp:
bpl (multdim.EXISTS (my_3d_array, 1, 5, 800));
bpl (multdim.EXISTS (my_3d_array, 6000, 5, 800));
bpl (multdim.EXISTS (my_3d_array, 6000, 5, 807));
*/
  DBMS_OUTPUT.PUT_LINE (my_3d_array.COUNT);
END;
```

Файл `multdim2.pkg` на сайте книги содержит доработанную версию пакета `multdim`, в которой реализована возможность получения «срезов» трехмерной коллекции (одно измерение фиксируется, а из пространства выделяется двумерная плоскость, определяемая зафиксированным измерением). Например, срез температурного пространства даст мне диапазон температур по заданной широте или долготе.

Кроме трудностей, связанных с реализацией срезов, возникает интересный вопрос: будут ли чем-нибудь отличаться срезы по плоскости XY, XZ или YZ в этом симметричном кубе данных? Существенные различия могут повлиять на организацию многомерных коллекций.

Я рекомендую вам самостоятельно изучить эту тему и реализацию пакета `multdim2.pkg`.

Расширение `string_tracker` для многоуровневых коллекций

Рассмотрим другой пример применения многоуровневых коллекций: расширение пакета `string_tracker` из раздела, посвященного строковому индексированию, для поддержки множественных списков строк.

Пакет `string_tracker` удобен, но он позволяет отслеживать только один набор «используемых» строк в любой момент времени. А если мне потребуется отслеживать сразу несколько списков? Это очень легко делается при помощи многоуровневых коллекций:

```
/* Файл в Сети: string_tracker2.pks/pkb */
1 PACKAGE BODY string_tracker
2 IS
```

```

3  SUBTYPE maxvarchar2_t IS VARCHAR2 (32767);
4  SUBTYPE list_name_t IS maxvarchar2_t;
5  SUBTYPE variable_name_t IS maxvarchar2_t;
6
7  TYPE used_aat IS TABLE OF BOOLEAN INDEX BY variable_name_t;
8
9  TYPE list_rt IS RECORD (
10     description      maxvarchar2_t
11     , list_of_values  used_aat
12 );
13
14 TYPE list_of_lists_aat IS TABLE OF list_rt INDEX BY list_name_t;
15
16 g_list_of_lists  list_of_lists_aat;
17
18 PROCEDURE create_list (
19     list_name_in      IN   list_name_t
20     , description_in  IN   VARCHAR2 DEFAULT NULL
21 )
22 IS
23 BEGIN
24     g_list_of_lists (list_name_in).description := description_in;
25 END create_list;
26
27 PROCEDURE mark_as_used (
28     list_name_in      IN   list_name_t
29     , variable_name_in IN   variable_name_t
30 )
31 IS
32 BEGIN
33     g_list_of_lists (list_name_in)
34         .list_of_values (variable_name_in) := TRUE;
35 END mark_as_used;
36
37 FUNCTION string_in_use (
38     list_name_in      IN   list_name_t
39     , variable_name_in IN   variable_name_t
40 )
41 RETURN BOOLEAN
42 IS
43 BEGIN
44     RETURN g_list_of_lists (list_name_in)
45         .list_of_values.EXISTS (variable_name_in);
46 EXCEPTION
47     WHEN NO_DATA_FOUND
48     THEN
49         RETURN FALSE;
50 END string_in_use;
51 END string_tracker;

```

В следующей таблице объясняются изменения, внесенные в пакет для поддержки многоуровневых коллекций.

| Строки | Описание |
|--------|--|
| 7 | И снова я создаю тип коллекции, индексируемой строками, для хранения строк пользователя |
| 9–12 | Затем я создаю запись для хранения всех атрибутов списка: описания и списка используемых строк в этом списке. Обратите внимание: имя списка не сохраняется как его атрибут. На первый взгляд это несколько странно, но имя списка является значением его индекса (см. далее) |
| 14–16 | В завершение создается тип многоуровневой коллекции: список списков, в котором каждый элемент коллекции верхнего уровня содержит запись, которая, в свою очередь, содержит коллекцию используемых строк |

продолжение ➤

| Строки | Описание |
|--------|--|
| 33–34 | Теперь процедура <code>mark_as_used</code> использует имя списка и имя переменной как индексы соответствующих коллекций: <code>g_list_of_lists (list_name_in)</code> <code>.list_of_values(variable_name_in) := TRUE;</code> Обратите внимание: если я помечаю имя переменной как используемое в новом списке, база данных создает для этого списка новый элемент в коллекции <code>g_list_of_lists</code> . Если поместить имя переменной как используемое в ранее созданном списке, то дело ограничивается добавлением нового элемента во вложенную коллекцию |
| 44–45 | Чтобы проверить, используется ли строка, мы смотрим, определено ли имя переменной как элемент в элементе списка списков: <code>RETURN g_list_of_lists (list_name_in)</code> <code>.list_of_values.EXISTS (variable_name_in);</code> |

Заметьте, что в третьей реализации `string_tracker` я использую именованные подтипы во всех объявлениях формальных параметров и особенно в секциях `INDEX BY` объявлений типов коллекций. Использование подтипов вместо жестко запрограммированных объявлений `VARCHAR2` способствует самодокументированию кода. Если этого не сделать, в один прекрасный день вы почешете в затылке и спросите себя: «А что, собственно, я использую как индекс этой коллекции?»

Максимальная глубина вложения

В процессе экспериментов с двух- и трехмерными массивами меня начал интересовать вопрос, до какой же глубины можно вкладывать эти многоуровневые коллекции. Чтобы получить ответ, я построил небольшой генератор кода, который позволяет передавать количество уровней вложения. Генератор строит процедуру, которая объявляет *N* типов коллекций, каждый из которых является таблицей с элементами типа предыдущей таблицы. Наконец, генератор присваивает значение строке, находящейся на полной глубине вложения коллекций.

Мне удалось создать коллекцию из по крайней мере 250 вложенных коллекций, прежде чем мой компьютер выдал ошибку памяти! Вряд ли какому-нибудь разработчику PL/SQL когда-либо потребуется такой уровень сложности. Если вы захотите провести подобный эксперимент в своей системе, найдите файл `gen_multcoll.sp` на сайте книги.

Работа с коллекциями в SQL

Я работаю с Oracle уже больше 22 лет, а с PL/SQL — более 18 лет, но мне еще не приходилось ломать голову над семантикой SQL так, как при первом знакомстве с *псевдофункциями коллекций*, появившимися в Oracle8. Основная цель псевдофункций — заставить таблицы баз данных работать как коллекции, и наоборот. Так как некоторые операции с данными лучше всего работают с данными, находящимися в некоторой конкретной форме, эти функции открывают программисту доступ к богатому и интересному набору структур и операций.



Псевдофункции коллекций не поддерживаются в PL/SQL — только в SQL. Впрочем, ничто не мешает использовать эти операторы в SQL-инструкциях, присутствующих в коде PL/SQL, поэтому в высшей степени полезно понимать, когда и как эти псевдофункции используются. Примеры приведены в следующих разделах.

Существуют четыре разновидности псевдофункций коллекций:

- `CAST` — отображает коллекцию одного типа на коллекцию другого типа. В частности, может использоваться для отображения `VARRAY` на вложенную таблицу.

- **COLLECT** — агрегирует данные в коллекцию в SQL. Эта функция, впервые появившаяся в Oracle Database 10g, была усовершенствована в 11.2 для поддержки упорядочения данных и устранения дубликатов.
- **MULTISET** — отображает таблицу базы данных на коллекцию. С псевдофункциями **MULTISET** и **CAST** появляется возможность выборки строки из таблицы базы данных как из столбца с типом коллекции.
- **TABLE** — отображает коллекцию на таблицу базы данных. Функция является обратной по отношению к **MULTISET**: она возвращает один столбец, содержащий отображаемую таблицу.

Псевдофункции были введены для работы с коллекциями, находящимися в базе данных. Коллекции играют важную роль в программах PL/SQL, и не в последнюю очередь из-за высочайшей эффективности перемещения данных между базой данных и приложением.

Да, в ходе изучения псевдофункций могут возникнуть проблемы. Но тех, кому доставляет удовольствие разбираться в нетривиальном коде, эти расширения SQL приведут в полный восторг.

Псевдофункция CAST

Оператор **CAST** может использоваться в командах SQL для преобразования одного встроенного типа данных или типа коллекции в другой встроенный тип данных или тип коллекции. Иначе говоря, в коде SQL конструкция **CAST** может использоваться вместо **TO_CHAR** для преобразования чисел в строки.

У **CAST** также имеется другое полезное применение — преобразование между типами коллекций. В следующем примере рассматривается преобразование именованной коллекции. Допустим, таблица `color_models` создана на основе типа **VARRAY**:

```
TYPE color_nt AS TABLE OF VARCHAR2(30)
```

```
TYPE color_vat AS VARRAY(16) OF VARCHAR2(30)
```

```
TABLE color_models (  
  model_type VARCHAR2(12),  
  colors color_vat);
```

Столбец `colors` можно преобразовать во вложенную таблицу с последующим применением к результату псевдофункции **TABLE** (см. далее). База данных присваивает единственному столбцу полученной виртуальной таблицы имя **COLUMN_VALUE**. Его можно заменить любым другим именем при помощи псевдонима столбца:

```
SELECT COLUMN_VALUE my_colors  
  FROM TABLE (SELECT CAST(colors AS color_nt)  
                FROM color_models  
                WHERE model_type = 'RGB')
```

CAST преобразует тип коллекции `color_vat` к типу `color_nt`. При этом **CAST** не может служить приемником для команд **INSERT**, **UPDATE** или **DELETE**.



Начиная с Oracle Database 10g явное преобразование коллекции в операторе **TABLE** уже не требуется. Вместо этого база данных автоматически определяет правильный тип.

Также возможно преобразование «группы строк таблицы» (например, результата подзапроса) к конкретному типу коллекции. Для решения этой задачи используется функция **MULTISET**, описанная в следующем разделе.

Псевдофункция COLLECT

Агрегатная функция COLLECT, появившаяся в Oracle 10g, позволяет объединить данные из команды SQL в коллекцию. В 11.2 эта функция была усовершенствована, и разработчик получил возможность упорядочить агрегированные результаты и устранить дубликаты в процессе агрегирования. Пример вызова COLLECT с упорядочением результатов (приводятся только первые три строки):

```
SQL> CREATE OR REPLACE TYPE strings_nt IS TABLE OF VARCHAR2 (100)
2 /
```

```
SQL> SELECT department_id,
2         CAST (COLLECT (last_name ORDER BY hire_date) AS strings_nt)
3         AS by_hire_date
4         FROM employees
5 GROUP BY department_id
6 /
```

```
DEPARTMENT_ID BY_HIRE_DATE
```

```
-----
10 STRINGS_NT('Whalen')
20 STRINGS_NT('Hartstein', 'Fay')
30 STRINGS_NT('Raphaely', 'Khoo', 'Tobias', 'Baida', 'Colmenares')
```

Следующий пример демонстрирует удаление дубликатов в процессе агрегирования:

```
SQL> SELECT department_id,
2         CAST (COLLECT (DISTINCT job_id) AS strings_nt)
3         AS unique_jobs
4         FROM employees
5 GROUP BY department_id
6 /
```

```
DEPARTMENT_ID UNIQUE_JOBS
```

```
-----
10 STRINGS_NT('AD_ASST')
20 STRINGS_NT('MK_MAN', 'MK_REP')
30 STRINGS_NT('PU_CLERK', 'PU_MAN')
```

Превосходная статья о COLLECT размещена на сайте Oracle Developer по адресу <http://www.oracle-developer.net/display.php?id=514>.

Псевдофункция MULTISSET

Функция MULTISSET может использоваться только в сочетании с CAST. MULTISSET позволяет получить набор данных и «на ходу» преобразовать его к типу коллекции. (Не путайте функцию SQL MULTISSET с операторами PL/SQL MULTISSET для вложенных таблиц, рассматриваемыми в разделе «Операции мультимножеств с вложенными таблицами».) Простейшая форма MULTISSET выглядит так:

```
SELECT CAST (MULTISSET (SELECT поле FROM таблица) AS тип_коллекции) FROM DUAL;
```

MULTISSET также может использоваться в ассоциированном подзапросе в списке выборки:

```
SELECT outerfield,
       CAST(MULTISSET(SELECT field FROM whateverTable
                       WHERE correlationCriteria)
          AS collectionTypeName)
FROM outerTable
```

Этот прием позволяет интерпретировать объединения так, словно они содержат коллекции. Допустим, имеется подчиненная таблица, в которой для каждой птицы из главной таблицы birds указаны страны, в которой эта птица живет:

```
CREATE TABLE birds (
  genus VARCHAR2(128),
  species VARCHAR2(128),
```

```

        colors color_tab_t,
        PRIMARY KEY (genus, species)
    );

CREATE TABLE bird_habitats (
    genus VARCHAR2(128),
    species VARCHAR2(128),
    country VARCHAR2(60),
    FOREIGN KEY (genus, species) REFERENCES birds (genus, species)
);

CREATE TYPE country_tab_t AS TABLE OF VARCHAR2(60);

```

Главная и подчиненная таблицы объединятся в инструкции `SELECT`, преобразующей подчиненные записи в коллекцию. Данная возможность чрезвычайно важна для программ «клиент/сервер», потому что она позволяет сократить количество циклов пересылки данных без затрат на дублирование главной записи в каждой подчиненной записи:

```

DECLARE
    CURSOR bird_curs IS
        SELECT b.genus, b.species,
               CAST(MULTISET(SELECT bh.country FROM bird_habitats bh
                               WHERE bh.genus = b.genus
                                   AND bh.species = b.species)
                   AS country_tab_t)
        FROM birds b;
    bird_row bird_curs%ROWTYPE;
BEGIN
    OPEN bird_curs;
    FETCH bird_curs INTO bird_row;
    CLOSE bird_curs;
END;

```

Как и псевдофункция `CAST`, `MULTISET` не может служить приемником для команд `INSERT`, `UPDATE` или `DELETE`.

Псевдофункция TABLE

Оператор `TABLE` преобразует столбец со значениями-коллекциями в источник, из которого можно получать данные инструкцией `SELECT`. Вроде бы сложно, но в этом разделе приведен пример, разобраться в котором будет легко.

Допустим, имеется таблица базы данных со столбцом, относящимся к типу коллекции. Как определить, какие строки таблицы содержат коллекцию, удовлетворяющую определенному критерию? Иначе говоря, как осуществить выборку из таблицы с условием `WHERE` для содержимого коллекции? Конечно, хотелось бы использовать запись следующего вида:

```

SELECT *
  FROM table_name
 WHERE collection_column
        HAS CONTENTS 'whatever';  -- НЕДОПУСТИМО! Такого синтаксиса нет!

```

Именно эту задачу можно решить при помощи функции `TABLE`. Вернемся к примеру с таблицей `color_models`; как получить список всех цветовых моделей, содержащих цвет `RED`? Вот как это делается:

```

SELECT *
  FROM color_models c
 WHERE 'RED' IN
        (SELECT * FROM TABLE(c.colors));

```

В `SQL*Plus` будет получен следующий результат:

```

MODEL_TYPE  COLORS
-----
RGB          COLOR_TAB_T('RED', 'GREEN', 'BLUE')

```

Приведенный запрос означает: «перебрать содержимое таблицы `color_models` и вернуть все записи, список цветов которых содержит как минимум один элемент `RED`». Если бы в таблице нашлись другие строки, в которых в столбце `colors` присутствует элемент `RED`, то они бы тоже были выведены в результирующем наборе `SQL*Plus`.

Как было показано выше, в единственном аргументе `TABLE` передается коллекция, в качестве которой может передаваться столбец, уточненный псевдонимом:

`TABLE(псевдоним.имя_коллекции)`

`TABLE` возвращает содержимое коллекции, преобразованное в виртуальную таблицу базы данных (и соответственно поддерживающую `SELECT`). Из этой таблицы можно получить данные командой `SELECT`, и с этими данными можно делать все то, что и с любыми другими: объединять их с другими наборами данных, выполнять операции множеств (`UNION`, `INTERSECT`, `MINUS` и т. д.). В приведенном примере она используется для выполнения подзапроса. Пример использования `TABLE` с локальной переменной `PL/SQL`:

```
/* Файл в Сети: nested_table_example.sql */
/* Создание типа уровня схемы. */
CREATE OR REPLACE TYPE list_of_names_t
  IS TABLE OF VARCHAR2 (100);
/
/* Заполнение коллекции с последующим использованием цикла FOR
с курсором для выборки всех элементов и их вывода. */
DECLARE
  happyfamily list_of_names_t := list_of_names_t ();
BEGIN
  happyfamily.EXTEND (6);
  happyfamily (1) := 'Eli';
  happyfamily (2) := 'Steven';
  happyfamily (3) := 'Chris';
  happyfamily (4) := 'Veve';
  happyfamily (5) := 'Lauren';
  happyfamily (6) := 'Loey';
  FOR rec IN ( SELECT COLUMN_VALUE family_name
               FROM TABLE (happyfamily)
               ORDER BY family_name)
  LOOP
    DBMS_OUTPUT.put_line (rec.family_name);
  END LOOP;
END;
```

До выхода Oracle Database 12c псевдофункция `TABLE` могла использоваться только с вложенными таблицами и массивами `VARRAY` и только в том случае, если их типы определялись на уровне схемы конструкцией `CREATE OR REPLACE TYPE` (у этого правила было одно исключение: конвейерные табличные функции могли работать с типами, определенными в спецификациях пакета). Однако начиная с Oracle Database 12c псевдофункция `TABLE` может использоваться с вложенными таблицами, массивами `VARRAY` и ассоциативными массивами с целочисленным индексированием при условии, что их типы определяются в спецификации пакета:

```
/* Файл в Сети: 12c_table_pf_with_aa.sql */
/* Создание типа на базе пакета. */
CREATE OR REPLACE PACKAGE aa_pkg
IS
  TYPE strings_t IS TABLE OF VARCHAR2 (100)
    INDEX BY PLS_INTEGER;
END;
/
/* Заполнение коллекции с последующим использованием цикла FOR
с курсором для выборки всех элементов и их вывода. */
```

```

DECLARE
  happyfamily aa_pkg.strings_t;
BEGIN
  happyfamily (1) := 'Me';
  happyfamily (2) := 'You';

  FOR rec IN ( SELECT COLUMN_VALUE family_name
                FROM TABLE (happyfamily)
                ORDER BY family_name)
  LOOP
    DBMS_OUTPUT.put_line (rec.family_name);
  END LOOP;
END;
/

```

Однако псевдофункция TABLE не может использоваться с локально объявленным типом коллекции, как видно из следующего примера (также обратите внимание на то, что сообщение об ошибке еще не было изменено в соответствии с расширенной областью применения TABLE):

```

/* Файл в Сети: 12c_table_pf_with_aa.sql */
DECLARE
  TYPE strings_t IS TABLE OF VARCHAR2 (100)
    INDEX BY PLS_INTEGER;

  happyfamily strings_t;
BEGIN
  happyfamily (1) := 'Me';
  happyfamily (2) := 'You';

  FOR rec IN ( SELECT COLUMN_VALUE family_name
                FROM TABLE (happyfamily)
                ORDER BY family_name)
  LOOP
    DBMS_OUTPUT.put_line (rec.family_name);
  END LOOP;
END;
/

```

```

ERROR at line 12:
ORA-06550: line 12, column 32:
PLS-00382: expression is of wrong type
ORA-06550: line 12, column 25:
PL/SQL: ORA-22905: cannot access rows from a non-nested table item

```

Еще раз повторим: псевдофункции коллекций недоступны в коде PL/SQL, но программисту PL/SQL определенно стоит научиться пользоваться ими в командах SQL! Псевдофункции (особенно TABLE) очень удобны при использовании *табличных функций*, появившихся в Oracle9i Database. Табличная функция возвращает коллекцию и может использоваться в секции FROM запроса. Эта возможность рассматривается в главе 17.

Сортировка содержимого коллекций

Одна из самых замечательных особенностей псевдофункций заключается в том, что они позволяют применять операции SQL к содержимому структур данных PL/SQL (по крайней мере к вложенным таблицам и VARRAY). Например, при помощи ORDER BY можно извлечь информацию из вложенной таблицы в нужном порядке. Сначала таблица базы данных заполняется именами писателей:

```

TYPE names_t AS TABLE OF VARCHAR2 (100)
TYPE authors_t AS TABLE OF VARCHAR2 (100)
TABLE favorite_authors (name varchar2(200));

```

```
BEGIN
  INSERT INTO favorite_authors VALUES ('Robert Harris');
  INSERT INTO favorite_authors VALUES ('Tom Segev');
  INSERT INTO favorite_authors VALUES ('Toni Morrison');
END;
```

А теперь мне хотелось бы объединить эту информацию с данными из программы PL/SQL:

```
DECLARE
  scifi_favorites  authors_t
    := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
BEGIN
  DBMS_OUTPUT.put_line ('I recommend that you read books by:');
  FOR rec IN (SELECT COLUMN_VALUE favs
              FROM TABLE (CAST (scifi_favorites AS  names_t))
              UNION
              SELECT NAME
              FROM favorite_authors)
  LOOP
    DBMS_OUTPUT.put_line (rec.favs);
  END LOOP;
END;
```

Обратите внимание на использование UNION для объединения данных из таблицы и коллекции. Этот прием также можно применить только к данным PL/SQL с целью сортировки:

```
DECLARE
  scifi_favorites  authors_t
    := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
BEGIN
  DBMS_OUTPUT.put_line ('I recommend that you read books by:');
  FOR rec IN (SELECT COLUMN_VALUE FavS
              FROM TABLE (CAST (scifi_favorites AS authors_t))
              ORDER BY COLUMN_VALUE)
  LOOP
    DBMS_OUTPUT.put_line (rec.favs);
  END LOOP;
END;
```



COLUMN_VALUE в приведенном выше запросе — сгенерированное системой имя столбца, созданного оператором TABLE (при выборке только одного столбца). Если запрос обращен к нескольким столбцам, вместо COLUMN_VALUE будут использованы имена атрибутов соответствующего объектного типа.

Операции мультимножеств с вложенными таблицами

Начиная с Oracle Database 10g в работу с коллекциями было внесено важное усовершенствование: база данных интерпретирует вложенные таблицы как мультимножества — причем даже в большей степени, чем они ими являются. База данных предоставляет высокоуровневые операции с множествами, которые могут применяться к вложенным таблицам (а на данный момент — *только* к вложенным таблицам). В таблице приводится краткая сводка таких операций.

| Операция | Возвращаемое значение | Описание |
|----------|-----------------------|--|
| = | BOOLEAN | Сравнивает две вложенные таблицы и возвращает TRUE, если совпадают имена их типов и мощность (кардинальное число) и они содержат равные элементы |

| Операция | Возвращаемое значение | Описание |
|-----------------------------------|-----------------------|--|
| <> или != | BOOLEAN | Сравнивает две вложенные таблицы и возвращает FALSE, если они отличаются по имени типа, мощности или равенству элементов |
| [NOT] IN () | BOOLEAN | Возвращает TRUE[FALSE], если вложенная таблица слева от IN существует в списке вложенных таблиц в круглых скобках |
| x MULTISET EXCEPT [DISTINCT] y | NESTED TABLE | Выполняет операцию вычитания множеств с вложенными таблицами x и y, возвращая вложенную таблицу, элементы которой присутствуют в x, но не в y. x, y и возвращаемая вложенная таблица должны относиться к одному типу. Ключевое слово DISTINCT приказывает Oracle удалить дубликаты из полученной вложенной таблицы |
| x MULTISET INTERSECT [DISTINCT] y | NESTED TABLE | Выполняет операцию пересечения множеств с вложенными таблицами x и y, возвращая вложенную таблицу, элементы которой присутствуют как в x, так и в y. x, y и возвращаемая вложенная таблица должны относиться к одному типу. Ключевое слово DISTINCT приказывает Oracle удалить дубликаты из полученной вложенной таблицы (включая дубликаты NULL, если они существуют) |
| x MULTISET UNION [DISTINCT] y | NESTED TABLE | Выполняет операцию объединения множеств с вложенными таблицами x и y, возвращая вложенную таблицу, элементы которой присутствуют в x или в y. x, y и возвращаемая вложенная таблица должны относиться к одному типу. Ключевое слово DISTINCT приказывает Oracle удалить дубликаты из полученной вложенной таблицы (включая дубликаты NULL, если они существуют) |
| SET(x) | NESTED TABLE | Возвращает вложенную таблицу x без дубликатов |
| x IS [NOT] A SET | BOOLEAN | Возвращает TRUE[FALSE], если вложенная таблица x состоит из уникальных элементов |
| x IS [NOT] EMPTY | BOOLEAN | Возвращает TRUE[FALSE], если вложенная таблица x пуста |
| e [NOT] MEMBER [OF] x | BOOLEAN | Возвращает TRUE[FALSE], если выражение e входит во вложенную таблицу x. Внимание: конструкция MEMBER в командах SQL работает крайне неэффективно, тогда как в PL/SQL ее производительность существенно выше |
| y [NOT] SUBMULTISET [OF] x | BOOLEAN | Возвращает TRUE[FALSE], если для каждого элемента в y существует соответствующий элемент в x |

В следующих разделах многие из этих операций будут рассмотрены более подробно. При этом я буду часто обращаться к следующему типу вложенной таблицы:

```
/* Файл в Сети: 10g_strings_nt.sql */
TYPE strings_nt IS TABLE OF VARCHAR2(100);
```

Я также буду неоднократно использовать следующий пакет:

```
/* Файл в Сети: 10g_authors.pkg */
CREATE OR REPLACE PACKAGE authors_pkg
IS
    steven_authors    strings_nt
        := strings_nt ('ROBIN HOBB'
            , 'ROBERT HARRIS'
            , 'DAVID BRIN'
            , 'SHERI S. TEPPER'
            , 'CHRISTOPHER ALEXANDER'
        );
    veva_authors      strings_nt
        := strings_nt ('ROBIN HOBB'
            , 'SHERI S. TEPPER'
            , 'ANNE MCCAFFREY'
        );
```

продолжение ➤

```

eli_authors strings_nt
:= strings_nt ( 'SHERI S. TEPPER'
               , 'DAVID BRIN'
               );

PROCEDURE show_authors (
    title_in IN VARCHAR2
  , authors_in IN strings_nt
)
;
END;
/

CREATE OR REPLACE PACKAGE BODY authors_pkg
IS
    PROCEDURE show_authors (
        title_in IN VARCHAR2
      , authors_in IN strings_nt
    )
    IS
    BEGIN
        DBMS_OUTPUT.put_line (title_in);

        FOR indx IN authors_in.FIRST .. authors_in.LAST
        LOOP
            DBMS_OUTPUT.put_line (indx || ' = ' || authors_in (indx));
        END LOOP;

        DBMS_OUTPUT.put_line ('_');
    END show_authors;
END;
/

```

Проверка равенства и принадлежности вложенных таблиц

До выхода Oracle Database 10g идентичность двух коллекций (то есть совпадение их содержимого) можно было проверить только одним способом: сравнить значения всех строк на равенство (а если коллекция содержит записи, то сравнить все поля каждой записи); пример кода приведен в файле 10g_coll_compare_old.sql. Начиная с Oracle Database 10g, со вложенными таблицами достаточно использовать стандартные операторы = и !=:

```

/* Файл в Сети: 10g_coll_compare.sql */
DECLARE
    TYPE clientele IS TABLE OF VARCHAR2 (64);
    group1 clientele := clientele ('Customer 1', 'Customer 2');
    group2 clientele := clientele ('Customer 1', 'Customer 3');
    group3 clientele := clientele ('Customer 3', 'Customer 1');
BEGIN
    IF group1 = group2
    THEN
        DBMS_OUTPUT.put_line ('Group 1 = Group 2');
    ELSE
        DBMS_OUTPUT.put_line ('Group 1 != Group 2');
    END IF;

    IF group2 != group3
    THEN
        DBMS_OUTPUT.put_line ('Group 2 != Group 3');
    ELSE
        DBMS_OUTPUT.put_line ('Group 2 = Group 3');
    END IF;
END;

```


Проверка равенства, реализованная для вложенных таблиц, рассматривает NULL по тем же правилам, что и для других операторов. NULL рассматривается как «неизвестная величина», то есть одно значение NULL никогда не равно другому значению NULL. Как следствие, если обе сравниваемые таблицы содержат значение NULL в одной строке, они *не будут* считаться равными.

Проверка принадлежности элемента вложенной таблице

Оператор MEMBER определяет, присутствует ли заданный элемент во вложенной таблице. Чтобы проверить, содержится ли вся вложенная таблица в другой вложенной таблице, используйте оператор SUBMULTISET. Пример:

```
/* Файл в Сети: 10g_submultiset.sql */
BEGIN
  bpl (authors_pkg.steven_authors
       SUBMULTISET OF authors_pkg.eli_authors
       , 'Father follows son?');
  bpl (authors_pkg.eli_authors
       SUBMULTISET OF authors_pkg.steven_authors
       , 'Son follows father?');
  bpl (authors_pkg.steven_authors
       NOT SUBMULTISET OF authors_pkg.eli_authors
       , 'Father doesn't follow son?');
  bpl (authors_pkg.eli_authors
       NOT SUBMULTISET OF authors_pkg.steven_authors
       , 'Son doesn't follow father?');
END;
```

Результаты выполнения этого кода:

```
SQL> @10g_submultiset
Father follows son? - FALSE
Son follows father? - TRUE
Father doesn't follow son? - TRUE
Son doesn't follow father? - FALSE
```

Высокоуровневые операции с множествами

Операции с множествами — такие, как UNION, INTERSECT и MINUS, — в высшей степени полезны и функциональны... именно потому, что они представляют очень простые и высокоуровневые концепции. Написание очень небольшого объема кода позволяет добиться замечательного эффекта. Возьмем следующий код, который демонстрирует применение различных операторов множеств:

```
/* Файл в Сети: 10g_union.sql */
1 DECLARE
2   our_authors strings_nt := strings_nt();
3 BEGIN
4   our_authors := authors_pkg.steven_authors
5                 MULTiset UNION authors_pkg.veva_authors;
6
7   authors_pkg.show_authors ('MINE then VEVA', our_authors);
8
9   our_authors := authors_pkg.veva_authors
10                 MULTiset UNION authors_pkg.steven_authors;
11
12  authors_pkg.show_authors ('VEVA then MINE', our_authors);
13
14  our_authors := authors_pkg.steven_authors
```

продолжение ➤

```

15             MULTISSET UNION DISTINCT authors_pkg.veva_authors;
16
17     authors_pkg.show_authors ('MINE then VEVA with DISTINCT', our_authors);
18
19     our_authors := authors_pkg.steven_authors
20             MULTISSET INTERSECT authors_pkg.veva_authors;
21
22     authors_pkg.show_authors ('IN COMMON', our_authors);
23
24     our_authors := authors_pkg.veva_authors
25             MULTISSET EXCEPT authors_pkg.steven_authors;
26
27     authors_pkg.show_authors (q'[ONLY VEVA'S]', our_authors);
28 END;
```

Результат, полученный при выполнении этого сценария:

```

SQL> @10g_union
MINE then VEVA
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ROBIN HOBB
7 = SHERI S. TEPPER
8 = ANNE MCCAFFREY

VEVA then MINE
1 = ROBIN HOBB
2 = SHERI S. TEPPER
3 = ANNE MCCAFFREY
4 = ROBIN HOBB
5 = ROBERT HARRIS
6 = DAVID BRIN
7 = SHERI S. TEPPER
8 = CHRISTOPHER ALEXANDER

MINE then VEVA with DISTINCT
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ANNE MCCAFFREY

IN COMMON
1 = ROBIN HOBB
2 = SHERI S. TEPPER

ONLY VEVA'S
1 = ANNE MCCAFFREY
```

Учтите, что оператор **MULTISSET UNION** работает не точно так же, как оператор **UNION** в SQL. Он не переупорядочивает данные и не удаляет дубликаты. Дубликаты вполне допустимы в мультимножествах. Если вы захотите удалить их, используйте **MULTISSET UNION DISTINCT**.

Обработка дубликатов во вложенной таблице

Итак, вложенная таблица может содержать дубликаты (значение, хранящееся в более чем одном экземпляре), причем эти дубликаты «переживут» даже операцию **MULTISSET UNION**. Иногда это именно то, что нужно; в других случаях требуется создать набор неповторяющихся значений. В Oracle предусмотрены следующие операторы:

- **Оператор SET** — преобразует набор элементов, содержащий дубликаты, во вложенную таблицу без дубликатов. Может рассматриваться как аналог `SELECT DISTINCT` для вложенных таблиц.
- **Операторы IS A SET и IS [NOT] A SET** — помогают получить ответы на вопросы вида: «Содержит ли вложенная таблица дубликаты?».

Эти возможности Oracle Database 10g и выше задействованы в следующем сценарии:

```
/* Файлы в Сети: 10g_set.sql, bpl2.sp */
BEGIN
  -- Add a duplicate author to Steven's list
  authors_pkg.steven_authors.EXTEND;
  authors_pkg.steven_authors(
    authors_pkg.steven_authors.LAST) := 'ROBERT HARRIS';

  distinct_authors :=
    SET (authors_pkg.steven_authors);

  authors_pkg.show_authors (
    'FULL SET', authors_pkg.steven_authors);

  bpl (authors_pkg.steven_authors IS A SET, 'My authors distinct?');
  bpl (authors_pkg.steven_authors IS NOT A SET, 'My authors NOT distinct?');
  DBMS_OUTPUT.PUT_LINE ('');

  authors_pkg.show_authors (
    'DISTINCT SET', distinct_authors);

  bpl (distinct_authors IS A SET, 'SET of authors distinct?');
  bpl (distinct_authors IS NOT A SET, 'SET of authors NOT distinct?');
  DBMS_OUTPUT.PUT_LINE ('');

END;
/
```

Результаты его выполнения:

```
SQL> @10g_set
FULL SET
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ROBERT HARRIS

My authors distinct? - FALSE
My authors NOT distinct? - TRUE

DISTINCT SET
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER

SET of authors distinct? - TRUE
SET of authors NOT distinct? - FALSE
```

Управление коллекциями уровня схемы

Информация, приведенная в этом разделе, не столь очевидна. Она поможет вам в работе с вложенными таблицами и `VARRAY`. Эти служебные аспекты не актуальны при работе с ассоциативными массивами.

Необходимые привилегии

Типы данных коллекций, находящиеся в базе данных, могут совместно использоваться несколькими пользователями. Как нетрудно догадаться, в совместном использовании задействованы привилегии. К счастью, ситуация не так сложна; к типам коллекций относится только одна привилегия Oracle — EXECUTE. Если вы, пользователь Scott, хотите разрешить пользователю Joe использовать тип color_tab_t в его программах, достаточно предоставить ему привилегию EXECUTE:

```
GRANT EXECUTE on color_tab_t TO JOE;
```

После этого Джо сможет использовать тип в синтаксисе *схема.тип*. Пример:

```
CREATE TABLE my_stuff_to_paint (
  which_stuff VARCHAR2(512),
  paint_mixture SCOTT.color_tab_t
)
NESTED TABLE paint_mixture STORE AS paint_mixture_st;
```

Привилегии EXECUTE также необходимы пользователям для запуска анонимных блоков PL/SQL, использующих объектный тип. Это одна из причин, по которым рекомендуется назначать имена модулям PL/SQL — пакетам, процедурам, функциям. Для таблиц, включающих столбцы коллекций, традиционные привилегии SELECT, INSERT, UPDATE и DELETE имеют смысл, пока не потребуются построить коллекцию для какого-либо из столбцов. Но если пользователь собирается выполнить команду INSERT или UPDATE для содержимого столбца коллекции, он должен иметь привилегию EXECUTE для типа, потому что это необходимо для использования конструктора по умолчанию.

Коллекции и словарь данных

База данных Oracle поддерживает несколько представлений словарей данных, которые предоставляют информацию о типах коллекций VARRAY и вложенных таблиц (табл. 12.4).

Таблица 12.4. Элементы словаря данных для типов коллекций

| Для получения ответа на вопрос... | ...используется представление... | как в следующем примере |
|--|----------------------------------|---|
| Какие типы коллекций я создал? | ALL_TYPES | SELECT type_name FROM all_types WHERE owner = USER AND typecode = 'COLLECTION'; |
| Как выглядело исходное определение типа коллекции Foo_t? | ALL_SOURCE | SELECT text FROM all_source WHERE owner = USER AND name = 'FOO_T' AND type = 'TYPE' ORDER BY line; |
| Какие столбцы реализуют Foo_t? | ALL_TAB_COLUMNS | SELECT table_name, column_name FROM all_tab_columns WHERE owner = USER AND data_type = 'FOO_T'; |
| Какие объекты базы данных зависят от Foo_t? | ALL_DEPENDENCIES | SELECT name, type FROM all_dependencies WHERE owner = USER AND referenced_name='FOO_T'; |

13

Другие типы данных

В этой главе рассматриваются встроенные типы данных PL/SQL, которые не были подробно описаны в предыдущих главах. Речь пойдет о типах `BOOLEAN`, `RAW` и `UROWID/ROWID`, а также о семействе типов `LOB` (Large Objects) — больших объектах. Кроме того, будет описано несколько важных предопределенных объектных типов, в том числе тип `XMLType`, обеспечивающий хранение данных в формате XML в столбцах таблиц; семейство типов `URI`, предназначенных для хранения информации `URI` (Uniform Resource Identifier), и семейство типов `Any`, предназначенных для хранения произвольных данных.

В Oracle11g изменилась терминология реализации `LOB`. Компания Oracle переработала реализацию `LOB` на основе технологии *SecureFiles*; старая технология `LOB`, использовавшаяся до выхода Oracle11g, сейчас называется *BasicFiles*. В этой главе мы также рассмотрим *SecureFiles* и преимущества в области производительности, которые обеспечивает эта обновленная технология.

Тип данных `BOOLEAN`

Логические значения и переменные чрезвычайно удобны. Так как логические переменные в PL/SQL могут принимать только три значения: `TRUE`, `FALSE` и `NULL`, их использование проясняет смысл программного кода. Сложное логическое выражение с множеством разных переменных заменяется одной логической переменной, имя которой напрямую выражает смысл выражения.

Приведем пример команды `IF` с использованием логической переменной (или функции) — только по этому короткому фрагменту трудно различить их):

```
IF report_requested
THEN
    print_report (report_id);
END IF;
```

Программная логика данного фрагмента не только делает его чуть более самодокументированным, но и способствует его защите от будущих изменений. Рассмотрим, к примеру, код реализации пользовательского интерфейса, который может предшествовать этой команде. Как программа определяет, нужно ли пользователю печатать отчет? Возможно, она выводит окно запроса с вариантами ответов «Да» и «Нет» или предлагает установить флажок или выбрать в списке нужный вариант. Для приведенного кода это

не имеет значения. Код реализации можно изменить в любой момент; если интерфейс будет правильно присваивать значение логической переменной `report_requested`, это не повлияет на работоспособность программы.



Логический тип данных поддерживается в языке PL/SQL, но не в базе данных Oracle. В PL/SQL можно создать логическую переменную и работать с ней, но создать таблицу с логическими столбцами вам не удастся.

Тот факт, что логическая переменная может принимать и значение `NULL`, отражается на структуре команды `IF...THEN...ELSE`. Для примера сравните две следующие команды:

```
IF report_requested
THEN
    NULL; -- Выполняется, если report_requested = TRUE
ELSE
    NULL; -- Выполняется, если report_requested = FALSE или IS NULL
END IF;
```

```
IF NOT report_requested
THEN
    NULL; -- Выполняется, если report_requested = FALSE
ELSE
    NULL; -- Выполняется, если report_requested = TRUE or IS NULL
END IF;
```

Если всем трем возможным значениям переменной должны соответствовать разные действия, используйте каскадную команду `IF`:

```
IF report_requested
THEN
    NULL; -- Выполняется, если report_requested = TRUE
ELSIF NOT report_requested
THEN
    NULL; -- Выполняется, если report_requested = FALSE
ELSE
    NULL; -- Выполняется, если report_requested IS NULL
END IF;
```

Подробнее о значениях `NULL` в командах `IF` рассказывается в главе 4.

Тип данных RAW

Тип данных `RAW` предназначен для хранения и обработки двоичных данных сравнительно небольшого объема. В отличие от `VARCHAR2` и других символьных типов, данные типа `RAW` никогда не преобразуются между наборами символов при передаче между базой данных и программой PL/SQL. Переменные типа `RAW` объявляются следующим образом:

имя_переменной `RAW`(*максимальный_размер*)

Значение *максимальный_размер* должно находиться в диапазоне от 1 до 32 767. Имейте в виду, что переменная PL/SQL типа `RAW` может содержать до 32 767 байт данных, а столбец базы данных этого же типа — не более 2000 байт.

Тип данных `RAW` используется редко. Как уже было сказано, он предназначен для хранения малых объемов двоичных данных. При работе с большими объемами двоичных файлов (графикой, звуковыми файлами и т. д.) следует использовать тип данных `BLOB` (Binary Large Object — большой двоичный объект). О нем будет рассказано далее в этой главе.

Типы данных UROWID и ROWID

Типы данных UROWID и ROWID предназначены для работы с идентификаторами строк базы данных. ROWID — *идентификатор строки* (ROW IDentifier), а точнее, двоичное значение, однозначно идентифицирующее строку данных в таблице Oracle, даже если таблица не имеет уникального ключа. Две записи, даже если они содержат одинаковые значения столбцов, обладают разными идентификаторами ROWID или UROWID.



Учтите, что значения ROWID в таблицах могут изменяться. В ранних версиях Oracle (Oracle8 и ранее) значения ROWIDs оставались неизменными на протяжении жизненного цикла строки. Но в версии Oracle8i были добавлены новые возможности, нарушающие это старое правило. Если для обычной или индексной таблицы разрешено перемещение строк, обновление может привести к изменению ROWID или UROWID строки. Кроме того, если с таблицей будет выполнена операция, из-за которой строка перейдет из одного физического блока данных в другой блок, значение ROWID строки изменится.

Впрочем, даже с учетом этого предупреждения значения ROWID приносят практическую пользу. Включение значений ROWID в командах SELECT, UPDATE, MERGE и DELETE в некоторых случаях повышает скорость обработки, поскольку обращение к строке по ее идентификатору выполняется быстрее, чем по первичному ключу. На рис. 13.1 использование ROWID в команде UPDATE сравнивается с использованием значений столбцов (например, первичного ключа).

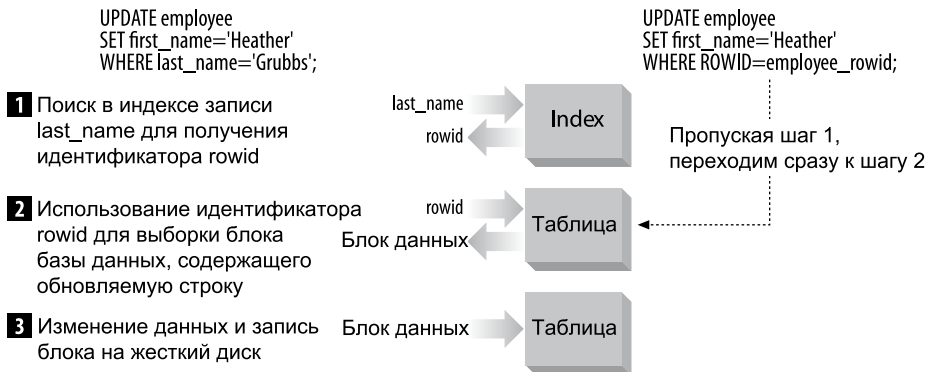


Рис. 13.1. Идентификатор ROWID ссылается непосредственно на строку таблицы

Исторически тип ROWID появился раньше типа UROWID. По мере добавления новых функциональных возможностей, таких как использование индекс-таблиц и шлюзов к другим базам данных, компания Oracle, естественно, разрабатывала и новые типы идентификаторов строк, и новые типы данных для их хранения. Так появился тип данных UROWID, используемый для хранения идентификаторов строк таблиц любого типа. Буква U в его названии означает «Universal» (*универсальный*), а переменная UROWID может содержать любое значение ROWID из любого типа таблиц.



Тип данных UROWID рекомендуется использовать во всех новых программах, работающих с идентификаторами строк. Тип ROWID обеспечивает обратную совместимость, но он не поддерживает все типы идентификаторов строк, используемых в современных базах данных Oracle. Тип UROWID надежнее — он поддерживает все типы ROWID, сохраняя все преимущества ускоренного доступа.

Получение идентификаторов строк

Чтобы получить ROWID для строки таблицы, добавьте ключевое слово в список выборки.

Пример:

```
DECLARE
    employee_rowid UROWID;
    employee_salary NUMBER;
BEGIN
    -- Выборка информации, которую мы собираемся модифицировать
    SELECT rowid, salary INTO employee_rowid, employee_salary
    FROM employees
    WHERE last_name='Grubbs' AND first_name='John';
END;
```

В терминологии Oracle ROWID называется «*псевдостолбцом*», потому что на самом деле столбца с именем ROWID в таблице не существует. Значение ROWID ближе к указателю — оно содержит физический адрес строки в таблице.

Использование идентификаторов строк

Преимущества использования ROWID проявляются при повторном обращении к строке, если оно производится часто или сопряжено со значительными затратами ресурсов. Вспомните пример из предыдущего раздела, когда мы извлекали из базы данных информацию об окладе работника. Допустим, нам потребовалось изменить величину оклада и ввести в базу данных новое значение. Конечно, для этого можно написать команду UPDATE с тем же условием WHERE, которое использовалось в команде SELECT:

```
DECLARE
    employee_rowid UROWID;
    employee_salary NUMBER;
BEGIN
    -- Выборка информации, которую мы собираемся модифицировать
    SELECT rowid, salary INTO employee_rowid, employee_salary
    FROM employees
    WHERE last_name='Grubbs' AND first_name='John';

    /* Вычисление нового оклада */

    UPDATE employees
    SET salary = employee_salary
    WHERE last_name='Grubbs' AND first_name='John';
END;
```

Конечно, этот код работает, но у него есть недостаток: необходимость повторения для UPDATE пути доступа, который уже использовался для SELECT. Скорее всего, поиск нужной записи потребовал обращения к одному-двум индексам. Но ведь программа уже обращалась к этим индексам для команды SELECT, так зачем выполнять всю работу дважды? Обращение к индексу производилось для получения ROWID с целью прямого обращения к записи. Включая значение ROWID в команду SELECT, я могу просто передать его команде UPDATE и обойтись без лишнего поиска по индексу:

```
DECLARE
    employee_rowid UROWID;
    employee_salary NUMBER;
BEGIN
    -- Выборка информации, которую мы собираемся модифицировать
    SELECT rowid, salary INTO employee_rowid, employee_salary
    FROM employees
    WHERE last_name='Grubbs' AND first_name='John';
```



```
/* Вычисление нового оклада */  
  
UPDATE employees  
  SET salary = employee_salary  
  WHERE rowid = employee_rowid;  
END;
```

Вспомните предупреждение о возможном изменении ROWID. Если в многопользовательской системе ROWID строки изменятся между командами SELECT и UPDATE, то код не будет работать так, как задумано. Почему? Потому что при разрешенном перемещении строк в стандартной таблице ROWID этой строки таблицы может измениться. А перемещение строк может быть разрешено потому, что администратор базы данных желает провести оперативную реорганизацию таблицы, или таблица может быть разбита на блоки, и перемещение строки позволит записи переместиться из одного блока в другой в процессе обновления.



Иногда для достижения аналогичного результата проще всего воспользоваться для выборки данных явным курсором, с последующей модификацией или удалением с применением конструкции WHERE CURRENT OF CURSOR. Подробнее данная возможность рассматривается в главе 15.

Конечно, использование ROWID ускоряет работу программ PL/SQL, потому что вы по сути опускаетесь на физический уровень управления базой данных. Однако хорошие приложения обычно не зависят от физической структуры данных. Они поручают управление физической структурой базе данных и административным программам, а сами ограничиваются логическим управлением данными. По этой причине использовать ROWID в приложениях обычно не рекомендуется.

Большие объекты данных

Oracle и PL/SQL поддерживают несколько разновидностей типов данных, предназначенных специально для работы с большими объектами (LOB, Large OBjects). Такие объекты позволяют хранить огромные (от 8 до 128 терабайт) объемы двоичных (например, графических) или текстовых данных.



До выхода Oracle9i Release2 в объектах LOB можно было хранить до 4 Гбайт данных. Начиная с Oracle10g, ограничение было повышено до величины от 8 до 128 терабайт (конкретное значение зависит от размера блока вашей базы данных).

В PL/SQL можно объявлять большие объекты четырех типов:

- BFILE — двоичный файл. Переменная этого типа содержит локатор файла, указывающий на файл операционной системы вне базы данных. Oracle интерпретирует содержимое файла как двоичные данные.
- BLOB — большой двоичный объект. Переменная этого типа содержит локатор LOB, указывающий на большой двоичный объект, хранящийся в базе данных.
- CLOB — большой символьный объект. Переменная этого типа содержит локатор LOB, указывающий на хранящийся в базе данных большой блок текстовых данных в наборе символов базы данных.
- NCLOB — большой символьный объект с поддержкой символов национальных языков (NLS). Переменная этого типа содержит локатор LOB, указывающий на хранящийся в базе данных большой блок текстовых данных с национальным набором символов.

Большие объекты можно разделить на две категории: *внутренние* и *внешние*. Внутренние большие объекты (типы BLOB, CLOB и NCLOB) хранятся в базе данных и могут участвовать в транзакциях на сервере базы данных. Внешние большие объекты (тип BFILE) представляют двоичные данные, хранящиеся в файлах операционной системы вне таблиц базы данных. Они не могут участвовать в транзакциях, то есть вносимые в них изменения нельзя сохранить или отменить в зависимости от результата транзакции. Целостность данных обеспечивается только на уровне файловой системы. Кроме того, повторное чтение из BFILE может приводить к разным результатам — в отличие от внутренних больших объектов, соответствующих модели логической целостности чтения.

LONG и LONG RAW

Вероятно, читатели, знакомые с Oracle, заметили, что мы до сих пор не упоминали о типах данных LONG и LONG RAW. И это не случайно. Конечно, в столбцах типа LONG и LONG RAW базы данных можно хранить большие объемы (до 2 Гбайт) соответственно символьных и двоичных данных. Однако максимальная длина соответствующих им переменных PL/SQL значительно меньше: всего лишь 32 760 байт, что даже меньше длины переменных VARCHAR2 и RAW (32 767 байт). С учетом столь странного ограничения в программах PL/SQL лучше использовать переменные типа VARCHAR2 и RAW, а не типа LONG и LONG RAW.

Значения типов LONG и LONG RAW, извлекаемые из базы данных и содержащие более 32 767 байт данных, не могут присваиваться переменным типа VARCHAR2 и RAW. Это крайне неудобное ограничение, из-за которого типы LONG и LONG RAW лучше вообще не применять.

Эти типы официально считаются устаревшими и поддерживаются только для сохранения обратной совместимости кода. Компания Oracle не рекомендует ими пользоваться, и я с ней полностью согласен. В новых приложениях вместо них лучше использовать типы CLOB и BLOB. А для существующих приложений в документации Oracle SecureFiles and Large Objects Developer's Guide приводятся рекомендации по преобразованию данных типа LONG в данные типа LOB.

Работа с большими объектами

Тема работы с большими объектами весьма объемна, поэтому мы не сможем рассмотреть все ее аспекты. Данный раздел следует рассматривать как введение в программирование больших объектов для разработчиков PL/SQL. Мы познакомимся с некоторыми нюансами, которые необходимо учитывать, и рассмотрим примеры важнейших операций. Хочется надеяться, что представленный материал станет хорошей основой для ваших дальнейших исследований в этой области.

Прежде чем переходить к основному материалу, необходимо указать, что все примеры данного раздела основаны на следующем определении таблицы (см. файл `ch13_code.sql` на сайте книги):

```
TABLE waterfalls (  
  falls_name VARCHAR2(80),  
  falls_photo BLOB,  
  falls_directions CLOB,  
  falls_description NCLOB,  
  falls_web_page BFILE)
```

Таблица содержит информацию о водопадах, расположенных в северной части штата Мичиган. На рис. 13.2 изображен водопад Драйер-Хоуз возле Мунисинга; в замерзшем состоянии его часто посещают альпинисты.

Каждый столбец таблицы предназначен для хранения больших объектов одного из четырех типов. Фотографии содержат большие объемы двоичных данных, поэтому для столбца `falls_photo column` определен тип `BLOB`. Столбцы с информацией о местоположении и с описанием водопадов (`falls_directions` и `falls_descriptions`) содержат текст, поэтому они определены соответственно с типами `CLOB` и `NCLOB`. В реальной таблице они имели бы один и тот же тип, но мы хотели продемонстрировать в своем примере все разновидности больших объектов. Наконец, копия веб-страницы каждого водопада хранится в HTML-файле вне базы данных; в таблицу включен столбец типа `BFILE`, содержащий указатели на эти файлы. Эти столбцы будут использоваться в последующих примерах для демонстрации различных аспектов работы с данными LOB в программах PL/SQL.

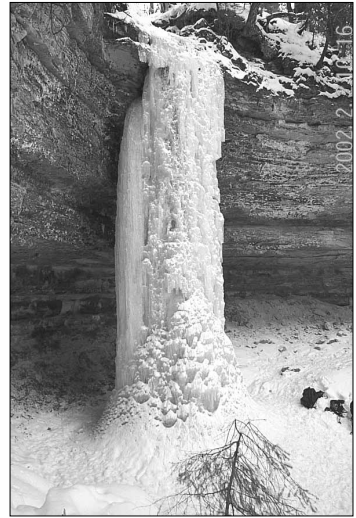


Рис. 13.2. Водопад Драйер-Хоуз возле Мунисинга (штат Мичиган)



При описании больших объектов мы будем часто использовать сокращение LOB для обозначения всех описываемых типов данных — `CLOB`, `BLOB`, `NCLOB` и `BFILE`. При обсуждении конкретного типа будет приводиться его полное название.

Понятие локатора LOB

В работе с LOB важнейшая роль отводится понятию *локатора* (locator) LOB. Локатором называется хранящийся в базе данных указатель на данные большого объекта. Давайте посмотрим, что же происходит при выборке данных из столбца типа `BLOB` в переменную PL/SQL того же типа:

```
DECLARE
  photo BLOB;
BEGIN
  SELECT falls_photo
    INTO photo
    FROM waterfalls
   WHERE falls_name='Dryer Hose';
```

Какое значение будет содержать переменная `photo` после выполнения команды `SELECT`? Графические данные изображения? Нет, в переменной будет храниться только указатель на двоичные данные. Результат выполнения команды `SELECT` показан на рис. 13.3.

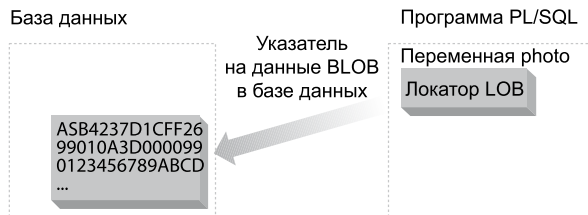


Рис. 13.3. Локатор LOB указывает на объект в базе данных

Этот механизм в корне отличается от того, как работают другие типы данных. Переменные и значения в столбцах LOB содержат локаторы больших объектов, которые идентифицируют реальные данные, хранящиеся в другом месте базы данных или вне ее. Для работы с данными типа LOB нужно сначала извлечь локатор, а затем с помощью встроенного пакета DBMS_LOB получить и/или модифицировать реальные данные. Так, для получения двоичных данных фотографии, локатор которой хранится в столбце BLOB из приведенной ранее таблицы, необходимо выполнить следующие действия:

1. Выполнить команду SELECT и выбрать из базы данных LOB-локатор фотографии.
2. Открыть объект LOB с помощью функции DBMS_LOB.OPEN.
3. Вызвать функцию DBMS_LOB.GETCHUNKSIZE для получения оптимального (с точки зрения затрат на выполнение операции чтения или записи) размера фрагмента данных типа LOB.
4. Вызвать функцию DBMS_LOB.GETLENGTH для определения количества байтов или символов объекта LOB.
5. Несколько раз вызвать функцию DBMS_LOB.GETLENGTH для считывания данных типа LOB.
6. Закрыть объект LOB.

Не все эти действия являются обязательными; не огорчайтесь, если что-то пока остается непонятным. Далее все эти операции будут описаны более подробно.

На первый взгляд кажется, что локаторы значительно усложняют схему работы с большими объектами. Однако предлагаемый Oracle подход имеет существенные преимущества, потому что он избавляет от необходимости возвращать все данные объекта LOB при каждой выборке строки из таблицы. Представьте, сколько времени потребовалось бы для передачи 128 терабайт данных LOB. Представьте, насколько это неэффективно, если вам нужна лишь небольшая часть этих данных. В механизме Oracle производится выборка локатора (быстрая операция), после чего читаются только те данные LOB, которые вам нужны. Кроме того, объекты LOB по умолчанию не кэшируются в буферном кэше, и для них не генерируются данные отмены (хотя генерируются данные повторного выполнения, если не установить параметр NOLOGGING). Таким образом, загрузка 50 гигабайт данных LOB не приведет к переполнению буферного кэша или таблиц отмены с общим снижением быстродействия. Раздельное кэширование и управление отменой LOB было дополнительно усовершенствовано в Oracle 11g... впрочем, об этом позднее.

LOB-ДАННЫЕ В ДОКУМЕНТАЦИИ ORACLE

Если вам часто приходится работать с большими объектами, настоятельно рекомендуем ознакомиться со следующими документами Oracle:

- SecureFiles and Large Objects Developer's Guide — руководство по программированию с использованием LOB для Oracle 11g и выше.
- Application Developer's Guide — Large Objects — руководство по программированию с использованием LOB для Oracle 10g и более ранних версий.
- PL/SQL Packages and Types Reference — глава с описанием пакета DBMS_LOB.
- SQL Reference — раздел «Datatypes» главы 2 содержит важную информацию о LOB-объектах.

Это далеко не полный список документации об объектах LOB, но в этих документах можно найти все наиболее важные сведения.

Большие объекты — пустые и равные NULL

Теперь, когда вы понимаете различие между локатором LOB и значением, на которое он ссылается, мы введем еще одно важное понятие: *пустой* объект LOB. Так называется локатор, не указывающий ни на какие данные. Это не то же самое, что значение NULL, то есть LOB-столбец (или переменная), не содержащий локатора LOB. Рассмотрим пример:

```
DECLARE
  directions CLOB;
BEGIN
  IF directions IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('directions is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('directions is not NULL');
  END IF;
END;
```

directions is NULL

Объявленная нами переменная CLOB содержит NULL, потому что ей пока не присвоено никакое значение. Выглядит все привычно, не правда ли? Да и происходит то же самое, что при работе с данными любого другого типа: переменная объявлена без присваивания значения, и ее проверка на NULL конструкцией IS NULL дает результат TRUE. В этом отношении данные LOB напоминают объекты: перед добавлением данных они тоже должны быть инициализированы. За дополнительной информацией об объектах обращайтесь к главе 26.

Давайте сделаем следующий шаг и инициализируем LOB. В следующем коде вызов EMPTY_CLOB инициализирует (но еще не заполняет!) переменную LOB. Сначала программный код:

```
DECLARE
  directions CLOB;
BEGIN
  IF directions IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Сначала directions is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Сначала directions is not NULL');
  END IF;
  DBMS_OUTPUT.PUT_LINE('Длина = '
    || DBMS_LOB.GETLENGTH(directions));

  -- инициализация переменной LOB
  directions := EMPTY_CLOB();

  IF directions IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('После инициализации directions is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('После инициализации directions is not NULL');
  END IF;
  DBMS_OUTPUT.PUT_LINE('Длина = '
    || DBMS_LOB.GETLENGTH(directions));
END;
```

Результат:

```
Сначала directions is NULL
Длина =
После инициализации directions is not NULL
Длина = 0
```

Мы видим, что переменная CLOB сначала представляет собой атомарное значение NULL. Не удивительно, что длина NULL-LOB тоже равна NULL. После инициализации переменной

типа CLOB встроенной функцией `EMPTY_CLOB` переменная уже не равна `NULL`, потому что она содержит значение: локатор. Функция `DBMS_LOB.GETLENGTH` показывает, что даже инициализированная (`NOT NULL`) переменная CLOB остается пустой.

Очень важно понимать это различие, потому что у типов LOB способ проверки наличия либо отсутствия данных получается более сложным, чем у скалярных типов. Для традиционных скалярных типов достаточно простой проверки `IS NULL`:

```
IF some_number IS NULL THEN
    -- Значит, данных нет
```

Если проверка `IS NULL` для `NUMBER` или `VARCHAR2` (или любого другого скалярного типа) возвращает `TRUE`, мы знаем, что переменная не содержит данных. Но при работе с LOB необходимо проверить не только равенство `NULL` (отсутствие локатора), но и длину:

```
IF some_clob IS NULL THEN
    -- Данных нет
ELSEIF DBMS_LOB.GETLENGTH(some_clob) = 0 THEN
    -- Данных нет
ELSE
    -- И только теперь данные присутствуют
END IF;
```

Итак, проверять нужно два условия вместо одного.



Для создания пустого объекта BLOB используется функция `EMPTY_BLOB()`. Для типов CLOB и NCLOB используется функция `EMPTY_CLOB()`.

Запись данных в объекты LOB

При наличии локатора данные записываются в LOB-объект с помощью одной из процедур пакета `DBMS_LOB`:

- `DBMS_LOB.WRITE` — запись в произвольную позицию.
- `DBMS_LOB.WRITEAPPEND` — присоединение данных в конец LOB.

В приведенном ниже коде (это расширенная версия примера, рассмотренного в предыдущих разделах главы) сначала создается локатор LOB для столбца `directions` таблицы `waterfalls`, а затем функция `DBMS_LOB.WRITE` записывает данные в CLOB-столбец строки «Munising Falls». Функция `DBMS_LOB.WRITEAPPEND` завершает дело:

```
/* Файл в Сети: munising_falls_01.sql */
DECLARE
    directions CLOB;
    amount BINARY_INTEGER;
    offset INTEGER;
    first_direction VARCHAR2(100);
    more_directions VARCHAR2(500);
BEGIN
    -- Удаление всех существующих строк со значением 'Munising Falls',
    -- чтобы этот пример мог выполняться многократно.
    DELETE
        FROM waterfalls
        WHERE falls_name='Munising Falls';

    -- Вставка новой строки и вызов функции EMPTY_CLOB() для создания локатора LOB
    INSERT INTO waterfalls
        (falls_name, falls_directions)
    VALUES ('Munising Falls', EMPTY_CLOB());
```

```

-- Получение локатора LOB, созданного предыдущей инструкцией INSERT
SELECT falls_directions
  INTO directions
  FROM waterfalls
 WHERE falls_name='Munising Falls';

-- Открытие объекта LOB; делать это необязательно, но лучше аккуратно
-- открывать и закрывать LOB-объекты
DBMS_LOB.OPEN(directions, DBMS_LOB.LOB_READWRITE);

-- Вызов DBMS_LOB.WRITE начинает запись.
first_direction := 'Follow I-75 across the Mackinac Bridge.';
amount := LENGTH(first_direction); -- количество записываемых символов
offset := 1; -- запись ведется с первого символа CLOB
DBMS_LOB.WRITE(directions, amount, offset, first_direction);

-- Вызов DBMS_LOB.WRITEAPPEND записывает дополнительные данные
more_directions := ' Take US-2 west from St. Ignace to Blaney Park.'
|| ' Turn north on M-77 and drive to Seney.'
|| ' From Seney, take M-28 west to Munising.';
DBMS_LOB.WRITEAPPEND(directions,
                      LENGTH(more_directions), more_directions);

-- И еще блок дополнительных данных
more_directions := ' In front of the paper mill, turn right on H-58.'
|| ' Follow H-58 to Washington Street. Veer left onto'
|| ' Washington Street. You'll find the Munising'
|| ' Falls visitor center across from the hospital at'
|| ' the point where Washington Street becomes'
|| ' Sand Point Road.';
DBMS_LOB.WRITEAPPEND(directions,
                      LENGTH(more_directions), more_directions);

-- Закрываем объект LOB - работа выполнена.
DBMS_LOB.CLOSE(directions);
END;
```

В этом примере используются обе процедуры, `WRITE` и `WRITEAPPEND`. Впрочем, это сделано лишь в учебных целях — с таким же успехом можно было ограничиться `WRITEAPPEND`, поскольку объект изначально не содержит никаких данных. Обратите внимание на открытие и закрытие объекта `CLOB`. Процедуры `OPEN` и `CLOSE` лучше вызывать явно, особенно при использовании Oracle Text. В противном случае создаваемые Oracle Text индексы будут обновляться при каждом вызове `WRITE` или `WRITEAPPEND`, а не один раз при вызове `CLOSE`.



В разделе, посвященном типу данных `BFILE`, будет рассказано, как прочитать данные LOB прямо из внешнего файла операционной системы.

При записи данных в объекты LOB обновлять LOB-столбец в таблице не нужно, потому что локатор LOB остается неизменным. Мы не изменяли содержимое `falls_directions` (локатор) — вместо этого данные добавлялись в объект LOB, на который ссылался локатор.

Обновление LOB происходит в контексте транзакции. В данном примере мы не сохраняли результаты транзакции инструкцией `COMMIT`. Однако если вы хотите, чтобы записанная в объект LOB информация была сохранена в базе данных, после выполнения блока команд PL/SQL нужно выполнить инструкцию `COMMIT`. Если же после блока команд PL/SQL будет выполнена инструкция `ROLLBACK`, все произведенные в этом блоке изменения данных будут отменены.

В нашем примере данные записываются в столбец типа CLOB. Данные BLOB обрабатываются аналогично, с той лишь разницей, что аргументы процедур WRITE и WRITEAPPEND имеют тип данных RAW, а не VARCHAR2.

Следующий пример, реализованный в SQL*Plus, выводит на экран данные, помещенные ранее в таблицу waterfalls. В следующем разделе будет показано, как извлекать данные с помощью разных процедур пакета DBMS_LOB:

```
SQL> SET LONG 2000
SQL> COLUMN falls_directions WORD_WRAPPED FORMAT A70
SQL> SELECT falls_directions
  2 FROM waterfalls
  3 WHERE falls_name='Munising Falls';
  4 /
```

FALLS_DIRECTIONS

```
-----
Follow I-75 across the Mackinac Bridge. Take US-2 west from St. Ignace
to Blaney Park. Turn north on M-77 and drive to Seney. From Seney,
take M-28 west to Munising. In front of the paper mill, turn right on
H-58. Follow H-58 to Washington Street. Veer left onto Washington
Street. You'll find the Munising Falls visitor center across from the
hospital at the point where Washington Street becomes Sand Point Road.
```

Чтение данных из объектов LOB

Данные читаются из LOB при помощи процедуры DBMS_LOB.READ. Но сначала, конечно, следует получить локатор LOB. При чтении данных типа CLOB указывается смещение, задаваемое в символах. Операция чтения начинается с позиции, заданной смещением, а первый символ CLOB всегда имеет номер 1. В случае применения типа данных BLOB смещение задается в байтах. Кроме того, в вызове процедуры DBMS_LOB.READ необходимо задать количество читаемых символов или байтов. Поскольку объекты LOB бывают очень большими, имеет смысл извлекать их содержимое по частям.

Рассмотрим фрагмент кода, который загружает и выводит данные о местонахождении водопада Munising Falls. Мы четко определяем количество читаемых символов, чтобы оно соответствовало ограничению на длину строки в пакете DBMS_OUTPUT, а выводимый текст выглядел аккуратно.

```
/* Файл в Сети: munising_falls_02.sql */
DECLARE
  directions CLOB;
  directions_1 VARCHAR2(300);
  directions_2 VARCHAR2(300);
  chars_read_1 BINARY_INTEGER;
  chars_read_2 BINARY_INTEGER;
  offset INTEGER;
BEGIN
  -- Получаем ранее добавленный локатор LOB
  SELECT falls_directions
    INTO directions
    FROM waterfalls
   WHERE falls_name='Munising Falls';

  -- Чтение начинается с первого символа
  offset := 1;

  -- Пытаемся прочитать 229 символов. В переменную chars_read_1
  -- будет записано реальное количество прочитанных символов
  chars_read_1 := 229;
  DBMS_LOB.READ(directions, chars_read_1, offset, directions_1);
```



```

-- Если прочитано 229 символов, информация о смещении обновляется
-- и производится попытка прочитать еще 255 символов.
IF chars_read_1 = 229 THEN
    offset := offset + chars_read_1;
    chars_read_2 := 255;
    DBMS_LOB.READ(directions, chars_read_2, offset, directions_2);
ELSE
    chars_read_2 := 0;
    directions_2 := '';
END IF;

-- Вывод общего количества прочитанных символов
DBMS_OUTPUT.PUT_LINE('Characters read = ' ||
    TO_CHAR(chars_read_1+chars_read_2));

-- вывод значения
DBMS_OUTPUT.PUT_LINE(directions_1);
DBMS_OUTPUT.PUT_LINE(directions_2);
END;
```

В результате выполнения этого кода выводится следующий текст:

```

Characters read = 414
Follow I-75 across the Mackinac Bridge. Take US-2 west from St. Ignace to Blaney
Park. Turn north on M-77 and drive to Seney. From Seney, take M-28 west to
Munising. In front of the paper mill, turn right on H-58. Follow H-58 to
Washington Street. Veer left onto Washington Street. You'll find the Munising
Falls visitor center across from the hospital at the point where Washington
Street becomes Sand Point Road.
```

Процедуре `DBMS_LOB.READ` в качестве второго параметра, объявленного как `IN OUT`, передается переменная `chars_read_1` (количество считанных символов). По окончании операции процедура обновляет его в соответствии с реальным количеством прочитанных символов. Если количество прочитанных символов (или байтов) будет меньше указанного, то это будет означать, что при чтении достигнут конец данных. К сожалению, обновить таким же образом информацию о смещении невозможно. Это приходится делать самостоятельно, по мере считывания последовательных фрагментов объектов `LOB`, в зависимости от количества прочитанных символов.



Для определения размера большого объекта можно воспользоваться функцией `DBMS_LOB.GET_LENGTH` (локатор `_lob`). Возвращаемое значение равно количеству байтов объекта `BLOB` или `BFILE` либо количеству символов объекта `CLOB`.

Особенности типа **BFILE**

Как упоминалось ранее, типы данных `BLOB`, `CLOB` и `NCLOB` представляют *внутренние* большие объекты, хранящиеся в базе данных, в то время как `BFILE` является *внешним* типом. Между объектами `BFILE` и внутренними `LOB` существуют три важных различия:

- Значение `BFILE` хранится не в базе данных, а в файле операционной системы.
- Объекты `BFILE` не участвуют в транзакциях (то есть внесенные в них изменения нельзя ни отменить, ни сохранить). Однако изменения, вносимые в локатор `BFILE`, в транзакциях можно и отменять, и сохранять.
- И в `PL/SQL`, и в `Oracle` объекты `BFILE` можно только считывать. Запись данных через типа `BFILE` не поддерживается. Внешние файлы, на которые указывают локаторы `BFILE`, должны создаваться вне СУБД `Oracle`.

Работая с данными типа `BFILE` в `PL/SQL`, вы работаете с локатором `LOB`, который просто указывает на хранящийся на сервере файл. Поэтому в разных строках таблицы базы

данных, содержащей столбец типа **BFILE**, могут храниться значения, указывающие на один и тот же файл.

Локаатор **BFILE** состоит из псевдонима каталога и имени файла. Для получения локаатора на основании этих двух параметров используется функция **BFILENAME**. *Псевдоним каталога* представляет собой определенное в Oracle имя, соответствующее имени каталога операционной системы. Псевдонимы позволяют программам PL/SQL работать с каталогами независимо от операционной системы. Имея привилегии **CREATE ANY DIRECTORY**, вы можете создать псевдоним каталога (сам каталог уже должен существовать в файловой системе) и предоставить доступ к нему следующим образом:

```
CREATE DIRECTORY bfile_data AS 'c:\PLSQL Book\Ch13_Misc_Datatypes\'

GRANT READ ON DIRECTORY bfile_data TO gennick;
```

Впрочем, создание псевдонимов каталогов и управление доступом к ним — это функции администратора базы данных, не имеющие прямого отношения к PL/SQL, поэтому я не буду подробно рассматривать их. Если вам понадобятся более подробные сведения, поговорите с администратором базы данных или ознакомьтесь с разделом документации Oracle *SQL Reference Manual*, посвященным команде **CREATE DIRECTORY**. Чтобы получить список каталогов, к которым у вас имеется доступ, обратитесь с запросом к представлению **ALL_DIRECTORIES**.

Создание локаатора BFILE

Чтобы создать локаатор **BFILE**, следует вызвать функцию **BFILENAME** и передать ей псевдоним каталога и имя файла. В следующем примере создается локаатор **BFILE** для HTML-файла, который затем сохраняется в таблице **waterfalls**.

```
DECLARE
    web_page BFILE;
BEGIN
    -- Удаление строки Tannery Falls, чтобы этот пример
    -- мог выполняться многократно.
    DELETE FROM waterfalls WHERE falls_name='Tannery Falls';

    -- Вызов BFILENAME для создания локаатора
    web_page := BFILENAME('BFILE_DATA','Tannery_Falls.htm');

    -- Сохранение локаатора в таблице waterfalls
    INSERT INTO waterfalls (falls_name, falls_web_page)
        VALUES ('Tannery Falls',web_page);
END;
```

Локаатор **BFILE** — всего лишь комбинация псевдонима каталога и имени файла. Реальный файл и каталог даже не обязаны существовать. Иначе говоря, Oracle позволяет создавать псевдонимы для несуществующих каталогов, а **BFILENAME** — локаторы для несуществующих файлов. Иногда это бывает удобно.



В имени каталога, которое вы указываете в вызовах **BFILENAME**, различается регистр символов, который должен соответствовать регистру, отображаемому представлением словаря данных **ALL_DIRECTORIES**. Сначала я использовал в своем примере запись **bfile_data** в нижнем регистре — и был ошеломлен многочисленными ошибками при попытке обратиться к внешнему файлу **BFILE data** (см. следующий раздел). В большинстве случаев имя каталога при вызове **BFILENAME** должно записываться в верхнем регистре.

Доступ к данным BFILE

Получив локатор **BFILE**, вы можете считывать данные из внешнего файла практически так же, как из объекта **BLOB**. В следующем примере из веб-страницы извлекаются первые 60 байт HTML-кода. Полученные данные, имеющие тип **RAW**, преобразуются в символьную строку функцией **UTL_RAW.CAST_TO_VARCHAR2**.

```
DECLARE
    web_page BFILE;
    html RAW(60);
    amount BINARY_INTEGER := 60;
    offset INTEGER := 1;
BEGIN
    -- Получение LOB-локатора веб-страницы
    SELECT falls_web_page
        INTO web_page
        FROM waterfalls
        WHERE falls_name='Tannery Falls';

    -- Открываем локатор, читаем 60 байт и закрываем локатор
    DBMS_LOB.OPEN(web_page);
    DBMS_LOB.READ(web_page, amount, offset, html);
    DBMS_LOB.CLOSE(web_page);

    -- Чтобы результат выводился в шестнадцатеричной форме,
    -- удалите символы комментария из следующей строки
    --DBMS_OUTPUT.PUT_LINE(RAWTOHEX(html));

    -- Преобразование данных типа RAW в читабельную символьную строку
    DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(html));
END;
```

Результат выполнения этого кода:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN
```

Максимальное число одновременно открытых в сеансе объектов **BFILE** устанавливается параметром базы данных **SESSION_MAX_OPEN_FILES**. Этот параметр определяет верхнюю границу количества файлов, которые могут быть одновременно открыты в сеансе (не только **BFILE**, но и любых других файлов, включая открытые с применением программ из пакета **UTL_FILE**).

Напомним, что Oracle поддерживает только чтение данных **BFILE**. Тип **BFILE** идеально подходит для работы с двоичными данными, созданными вне СУБД (графикой и т. д.). Например, вы можете загрузить на сервер изображения, отснятые цифровой камерой, создать локаторы **BFILE** для этих файлов и работать с ними в программах PL/SQL.

Использование BFILE для загрузки столбцов LOB

Тип **BFILE** не только обеспечивает доступ к двоичным файлам, созданным вне базы данных Oracle, но и является удобным средством загрузки внешних данных в столбцы **LOB**. В Oracle9i Release 1 для чтения двоичных данных из объекта **BFILE** и их записи в столбец **BLOB** использовалась функция **DBMS_LOB.LOADFROMFILE**, а в Oracle9i Release 2 появились намного более удобные средства:

- Функция **DBMS_LOB.LOADCLOBFROMFILE** загружает в столбец типа **CLOB** данные из объекта **BFILE** с необходимым преобразованием символов.
- Функция **DBMS_LOB.LOADBLOBFROMFILE** загружает в столбец типа **BLOB** данные из объекта **BFILE**. Она делает то же, что **DBMS_LOB.LOADFROMFILE**, но ее интерфейс приближен к интерфейсу функции **DBMS_LOB.LOADCLOBFROMFILE**.

Представьте, что информация о местонахождении водопада Tannery Falls хранится во внешнем текстовом файле с именем TanneryFalls.directions в каталоге, на который ссылается псевдоним BFILE_DATA. Следующий пример демонстрирует использование DBMS_LOB.LOADCLOBFROMFILE для загрузки информации в CLOB-столбец falls_directions таблицы waterfalls:

```
/* Файл в Сети: munising_falls_03.sql */
DECLARE
  Tannery_Falls_Directions BFILE
    := BFILENAME('BFILE_DATA', 'TanneryFalls.directions');
  directions CLOB;
  destination_offset INTEGER := 1;
  source_offset INTEGER := 1;
  language_context INTEGER := DBMS_LOB.default_lang_ctx;
  warning_message INTEGER;
BEGIN
  -- Удаление строки 'Tannery Falls',
  -- чтобы этот пример мог выполняться многократно.
  DELETE FROM waterfalls WHERE falls_name='Tannery Falls';

  -- Вставка новой строки с созданием локатора LOB
  -- вызовом EMPTY_CLOB()
  INSERT INTO waterfalls
    (falls_name, falls_directions)
    VALUES ('Tannery Falls', EMPTY_CLOB());

  -- Чтение локатора LOB, созданного предыдущей командой INSERT
  SELECT falls_directions
    INTO directions
    FROM waterfalls
    WHERE falls_name='Tannery Falls';

  -- Открытие объекта CLOB и исходного объекта BFILE
  DBMS_LOB.OPEN(directions, DBMS_LOB.LOB_READWRITE);
  DBMS_LOB.OPEN(Tannery_Falls_Directions);

  -- Загрузка содержимого BFILE в столбец CLOB
  DBMS_LOB.LOADCLOBFROMFILE
    (directions, Tannery_Falls_Directions,
     DBMS_LOB.LOBMAXSIZE,
     destination_offset, source_offset,
     NLS_CHARSET_ID('US7ASCII'),
     language_context, warning_message);

  -- Проверка единственно возможного предупреждения
  IF warning_message = DBMS_LOB.WARN_INCONVERTIBLE_CHAR THEN
    DBMS_OUTPUT.PUT_LINE (
      'Warning! Some characters couldn''t be converted.');
```

```
END IF;

-- Закрытие обоих объектов LOB
DBMS_LOB.CLOSE(directions);
DBMS_LOB.CLOSE(Tannery_Falls_Directions);
END;
```

Основная работа в этом фрагменте кода выполняется вызовом DBMS_LOB.LOADCLOBFROMFILE. Эта процедура читает данные из внешнего файла, выполняет все необходимые преобразования набора символов и записывает данные в столбец CLOB. Я использую константу DBMS_LOB.LOBMAXSIZE для определения объема загружаемых данных. Мне нужны *все* данные из внешнего файла, а DBMS_LOB.LOBMAXSIZE определяет максимальный объем, который может храниться в CLOB.

Смещения источника и приемника начинаются с 1. Чтение должно начинаться с первого символа из **BFILE**, а запись — с первого символа **CLOB**. Чтобы упростить множественные последовательные вызовы **LOADCLOBFROMFILE**, процедура обновляет оба смещения так, чтобы они указывали в следующую позицию за последним прочитанным символом.

Так как они являются параметрами **IN OUT**, при вызове процедуры должны использоваться переменные, а не константы.

Вызов **NLS_CHARSET_ID** возвращает идентификатор набора символов внешнего файла. Процедура **LOADCLOBFROMFILE** преобразует загружаемые данные из этого набора в набор символов базы данных. Единственное возможное предупреждение, которое может вернуть **LOADCLOBFROMFILE**, означает, что некоторые символы не могут быть преобразованы из исходного набора данных в итоговый. В программе это предупреждение проверяется в команде **IF**, следующей за загрузкой.



Не путайте предупреждения с ошибками **PL/SQL**; загрузка все равно будет выполнена в соответствии с запросом.

Следующий пример **SQL*Plus** демонстрирует загрузку данных из внешнего файла с использованием **LOADCLOBFROMFILE**:

```
SQL> SET LONG 2000
SQL> COLUMN falls_directions WORD_WRAPPED FORMAT A70
SQL> SELECT falls_directions
       2 FROM waterfalls
       3 WHERE falls_name='Tannery Falls';
       4 /
```

FALLS_DIRECTIONS

From downtown Munising, take Munising Avenue east. It will shortly turn into H-58. Watch for Washington Street veering off to your left. At that intersection you'll see a wooden stairway going into the woods on your right. Go up that stairway and follow the trail to the falls. Do not park on H-58! You'll get a ticket. You can park on Nestor Street, which is just uphill from the stairway.

SecureFiles и BasicFiles

Технология **SecureFiles**, появившаяся в **Oracle11g**, обладает рядом преимуществ по сравнению со старыми реализациями **LOB**, теперь известными под общим названием **BasicFiles**. Эти преимущества в основном относятся к внутренней реализации и остаются прозрачными для программистов — в коде используются те же ключевые слова, тот же синтаксис и операции. Во внутренней реализации **SecureFiles** усовершенствованы многие аспекты управления **LOB**, включая формат хранения данных на диске, кэширование, блокировку, отмену и алгоритмы управления дисковым пространством. Обновленная технология заметно улучшает быстродействие и позволяет устранять дублирование, сжимать и шифровать объекты **LOB** посредством определения простых параметров. Кроме того, был введен новый уровень ведения журнала **FILESYSTEM_LIKE_LOGGING** в дополнение к существующим режимам **LOGGING** и **NOLOGGING**. В новом режиме фиксируются только изменения метаданных (по аналогии с тем, как это делается в журнальных файловых системах).

По данным тестирования Oracle, повышение быстродействия от использования SecureFiles составляет от 200 до 900%. В простом тесте с загрузкой файлов PDF на сервер Microsoft Windows я наблюдал снижение времени загрузки от 80 до 90% от 169 секунд до 20–30 секунд (в зависимости от конфигурации и количества загрузок). В системе x86 Linux выигрыш был более скромным. Возможно, в вашей ситуации цифры будут другими, но ускорение будет наверняка!

Чтобы использовать SecureFiles при работе с данными LOB, задайте параметру базы данных `DB_SECUREFILE` значение `PERMITTED` (используется по умолчанию). Кроме того, табличное пространство, в котором будут храниться данные LOB, должно использовать технологию ASSM (Automatic Segment Space Management).



В Oracle Database 12c технология SecureFiles используется по умолчанию; в Oracle Database 11g (Release 1 или Release 2) механизм хранения определяется параметром `DB_SECUREFILE`. Если задать ему значение `ALWAYS`, то по умолчанию для объектов LOB используется технология SecureFiles; в противном случае используется BasicFiles.

Если вы не уверены относительно базы данных, обратитесь за помощью к администратору.

Устранение дубликатов

В режиме устранения дубликатов база данных хранит только одну копию каждого объекта LOB. Для каждого объекта генерируется хеш-код, который сравнивается с хеш-кодами других объектов LOB в таблице или в ее разделе. Учтите, что устранение дубликатов не работает между разделами и подразделами.

Сжатие

В режиме сжатия SecureFiles база данных сжимает данные LOB как на диске, так и в памяти. Поддерживаются режимы `MEDIUM` (среднее сжатие, используется по умолчанию) и `HIGH` (высокое сжатие). Высокое сжатие занимает больше времени, но уменьшает размер данных LOB. Простые тесты с файлами PDF показали, что загрузка данных в режиме `HIGH` занимала примерно на 25% больше времени, чем в режиме `MEDIUM`.

Чтобы включить режим устранения дубликатов одновременно со сжатием, укажите соответствующие параметры в определении LOB:

```
TABLE waterfalls
(
  falls_name          VARCHAR2 (80)
, falls_photo         BLOB
, falls_directions    CLOB
, falls_description   NCLOB
, falls_web_page      BFILE
)
LOB (falls_photo) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_directions) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_description) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
```

При включении обоих режимов сначала выполняется устранение дубликатов, а затем сжатие.

Шифрование

Режим шифрования SecureFiles также включается в разделе LOB команды `CREATE TABLE`. Также можно выбрать алгоритм шифрования; в Oracle 11g поддерживаются алгоритмы 3DES168, AES128, AES192 (по умолчанию) и AES256. Допускается использование любых комбинаций режимов устранения дубликатов, сжатия и шифрования:

```
TABLE waterfalls
(
  falls_name          VARCHAR2 (80)
, falls_photo         BLOB
, falls_directions    CLOB
, falls_description   NCLOB
, falls_web_page      BFILE
)
LOB (falls_photo) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_directions) STORE AS SECUREFILE (ENCRYPT USING 'AES256')
LOB (falls_description) STORE AS SECUREFILE
  (ENCRYPT DEDUPLICATE COMPRESS HIGH
)
```

Если база данных не была настроена на применение прозрачного шифрования данных (TDE Transparent Data Encryption), шифрование LOB потребует двух подготовительных шагов. Сначала необходимо создать *бумажник* (wallet) для хранения главного ключа. Если бумажник должен храниться в стандартном каталоге (\$ORACLE_BASE/admin/\$ORACLE_SID/wallet), его создание и открытие можно совместить в одном шаге:

```
ALTER SYSTEM SET ENCRYPTION KEY AUTHENTICATED BY "My-secret!passc0de";
```

Если бумажник должен храниться в нестандартном каталоге, вам придется задать этот каталог в файле SQLNET.ORA. Так, чтобы бумажник хранился в каталоге /oracle/wallet, включите в файл SQLNET.ORA следующий фрагмент:

```
ENCRYPTION_WALLET_LOCATION=(SOURCE=(METHOD=file)
  (METHOD_DATA=(DIRECTORY=/oracle/wallet)))
```

Созданный бумажник необходимо открывать заново после каждого перезапуска экземпляра. Открытие и закрытие бумажника выполняется следующим образом:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN AUTHENTICATED BY "My-secret!passc0de";
-- Закрытие бумажника
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
```

Временные объекты LOB

До сих пор речь шла о долговременном хранении больших объемов неструктурированных данных за счет использования типов LOB. Применяемые для этого типы LOB называются *постоянными*. Однако многим приложениям нужны *временные LOB*, используемые в качестве локальных переменных и не хранящиеся в базе данных. В этом разделе рассказывается о временных объектах LOB и о том, как работать с ними с помощью программ встроенного DBMS_LOB.

В Oracle8i и последующих версиях поддерживается возможность создания, освобождения, чтения и обновления временных объектов LOB через интерфейс OCI (Oracle Call Interface) и вызовы DBMS_LOB. По умолчанию временный объект LOB существует в течение всего сеанса, в котором он создан, но многие объекты явно освобождаются и до окончания сеанса. Временные объекты типа LOB являются удобной рабочей средой для быстрого выполнения различных операций с данными. Более высокая их производительность по сравнению с постоянными объектами LOB достигается за счет того, что эти операции не фиксируются в журнале и для них не генерируются записи отмены. При перезаписи или обновлении объекта LOB Oracle копирует весь объект в новый сегмент данных. Избавляясь от дополнительной нагрузки, приложение, выполняющее множество мелких операций с LOB, может работать значительно быстрее.

Только что созданный временный объект LOB всегда пуст; его локаторы не нужно (да и невозможно) инициализировать вызовами функций EMPTY_CLOB и EMPTY_BLOB. По умолчанию все временные объекты LOB удаляются в конце сеанса, в котором они

созданы. Если процесс неожиданно завершится аварийно или произойдет сбой базы данных, временные объекты LOB будут удалены, а занимаемая ими память освободится.

Временные объекты LOB, как и постоянные, хранятся на диске в базе данных. Термин «временные» не должен вводить вас в заблуждение. Временные объекты LOB записываются на диск, но не связываются с определенными столбцами таблиц, располагаясь во временном табличном пространстве сеанса. Следовательно, для их использования нужно обеспечить наличие достаточного объема временного табличного пространства.

Процесс создания и освобождения временных объектов типа LOB будет рассмотрен ниже. Затем вы узнаете, как определить, на временный или постоянный объект указывает локатор LOB. А напоследок мы расскажем о некоторых приемах администрирования, связанных с обработкой временных объектов.

Создание временного объекта LOB

Чтобы получить экземпляр временного объекта, необходимо сначала создать его. Один из способов создания основан на вызове процедуры `DBMS_LOB.CREATETEMPORARY`. Процедура создает временный объект типа BLOB или CLOB и соответствующий индекс во временном пространстве. Заголовок процедуры выглядит так:

```
DBMS_LOB.CREATETEMPORARY (
    lob_loc IN OUT NOCOPY [ BLOB | CLOB CHARACTER SET ANY_CS ],
    cache   IN BOOLEAN,
    dur      IN PLS_INTEGER := DBMS_LOB.SESSION);
```

Параметры `DBMS_LOB.CREATETEMPORARY` представлены в табл. 13.1.

Таблица 13.1. Параметры процедуры `CREATETEMPORARY`

| Параметр | Описание |
|----------|--|
| lob_loc | Локаатор LOB |
| cache | Указывает, должен ли объект LOB при чтении помещаться в буферный кэш |
| dur | Определяет время жизни LOB. Допустимыми значениями являются две именованные константы: <code>DBMS_LOB.SESSION</code> указывает, что временный объект LOB должен быть удален в конце сеанса (используется по умолчанию), а <code>DBMS_LOB.CALL</code> указывает, что он должен быть удален в конце текущего вызова программы, в которой он был создан |

Другой способ создания временного объекта LOB основан на объявлении переменной LOB в коде PL/SQL и присваивании ей значения. Например, следующий фрагмент создает временные объекты BLOB и CLOB:

```
DECLARE
    temp_clob CLOB;
    temp_blob BLOB;
BEGIN
    -- Присваивание значения пустой переменной CLOB или BLOB
    -- заставляет PL/SQL неявно создать временный объект LOB,
    -- существующий на протяжении сеанса.
    temp_clob := ' http://www.nps.gov/piro/';
    temp_blob := HEXTORAW('7A');
END;
```

Я не могу однозначно порекомендовать тот или иной вариант создания временных LOB, но вызов `DBMS_LOB.CREATETEMPORARY` более четко выражает намерения программиста.

Освобождение временного объекта LOB

Процедура `DBMS_LOB.FREETEMPORARY` освобождает временный объект типа BLOB или CLOB в текущем временном табличном пространстве. Заголовок процедуры:


```
PROCEDURE DBMS_LOB.FREETEMPORARY (
  lob_loc IN OUT NOCOPY
  [ BLOB | CLOB CHARACTER SET ANY_CS ]);
```

В следующем примере сначала создаются, а затем освобождаются два временных объекта LOB:

```
DECLARE
  temp_clob CLOB;
  temp_blob BLOB;
BEGIN
  -- В случае присваивания значения неинициализированной
  -- переменной CLOB или BLOB PL/SQL неявно создает временный объект,
  -- существующий только на время текущего сеанса.
  temp_clob := 'http://www.exploringthenorth.com/alger/alger.html';
  temp_blob := HEXTORAW('7A');

  DBMS_LOB.FREETEMPORARY(temp_clob);
  DBMS_LOB.FREETEMPORARY(temp_blob);
END;
```

После вызова `FREETEMPORARY` освобожденный локатор LOB (`lob_loc`) помечается как недействительный. Если присвоить его другому локатору с помощью обычного оператора присваивания PL/SQL, то и этот локатор будет освобожден и помечен как недействительный.



PL/SQL неявно освобождает временные объекты LOB при выходе за пределы области действия блока.

Проверка статуса объекта LOB

Функция `ISTEMPORARY` позволяет определить, является ли объект с заданным локатором временным или постоянным. Она возвращает целочисленный код: 1 для временного объекта, 0 для постоянного.

```
DBMS_LOB.ISTEMPORARY (
  lob_loc IN [ BLOB | CLOB CHARACTER SET ANY_CS ])
RETURN INTEGER;
```

Обратите внимание: функция возвращает числовой код (1 или 0), а не тип данных `BOOLEAN`.

Управление временными объектами LOB

Временные большие объекты обрабатываются не так, как обычные постоянные. Для них не поддерживаются транзакции, операции согласованного чтения, откаты и т. д. Такое сокращение функциональности имеет ряд следствий:

- Если при обработке временного объекта LOB произойдет ошибка, этот объект придется освобождать и начинать все заново.
- Одному временному объекту LOB не следует присваивать несколько локаторов. Отсутствие поддержки операций согласованного чтения и отката при использовании нескольких локаторов может привести к значительному снижению производительности.
- Если пользователь модифицирует временный объект LOB, на который указывает еще один локатор, создается *глубокая копия* данного объекта. После этого локаторы будут ссылаться на разные объекты. Директива компилятора `NOCOPY` запрещает глубокое копирование при передаче локаторов LOB в аргументах.

- Чтобы преобразовать временный объект в постоянный, нужно вызвать программу DBMS_LOB.COPY и скопировать временный объект LOB в постоянный. Временные локаторы LOB действуют только в рамках сеанса. Локатор нельзя передать из одного сеанса в другой. Если объект LOB должен использоваться в разных сеансах, сделайте его постоянным.

В Oracle9i Database появилось представление с именем V\$TEMPORARY_LOBS, которое содержит информацию о количестве существующих кэшированных и некешированных объектов LOB для сеанса. Администратор базы данных может объединить информацию V\$TEMPORARY_LOBS с представлением словаря данных DBA_SEGMENTS, чтобы узнать, сколько дискового пространства сеанс использует для хранения временных LOB.

Встроенные операции LOB

С первых дней появления функциональности LOB в Oracle многочисленные пользователи, программисты и разработчики запросов желали использовать LOB как очень большие разновидности обычных скалярных переменных. В частности, пользователи хотели работать с CLOB как с огромными строками, передавать их функциям SQL, использовать в условиях WHERE команд SQL и т. д. К их огорчению, объекты CLOB изначально не могли использоваться вместо VARCHAR2. Например, в Oracle8 Database и Oracle8i Database к столбцу CLOB нельзя было применить символьную функцию:

```
SELECT SUBSTR(falls_directions,1,60)
FROM waterfalls
```

Начиная с Oracle9i Database, объекты CLOB могут заменять VARCHAR2 во многих ситуациях:

- Объекты CLOB могут передаваться многим VARCHAR2-функциям PL/SQL и SQL — эти функции существуют в перегруженных версиях с параметрами VARCHAR2 и CLOB.
- В PL/SQL, но не в SQL, с переменными LOB могут использоваться различные операторы отношений, такие как «меньше» (<), «больше» (>) и «равно» (=).
- Значения CLOB можно присваивать переменным VARCHAR2, и наоборот. Также можно осуществлять выборку значений CLOB в переменные VARCHAR2, и наоборот. Такая возможность существует благодаря тому, что PL/SQL теперь выполняет неявные преобразования между типами CLOB и VARCHAR2.

Семантика SQL

Возможности, упомянутые в предыдущем разделе, Oracle называет «поддержкой семантики SQL для LOB». С точки зрения разработчика PL/SQL это означает, что с LOB можно работать на уровне встроенных операторов (вместо отдельного пакета).

Следующий пример демонстрирует некоторые возможности семантики SQL:

```
DECLARE
    name CLOB;
    name_upper CLOB;
    directions CLOB;
    blank_space VARCHAR2(1) := ' ';
BEGIN
    -- чтение VARCHAR2 в CLOB и применение функции к CLOB
    SELECT falls_name, SUBSTR(falls_directions,1,500)
    INTO name, directions
    FROM waterfalls
    WHERE falls_name = 'Munising Falls';

    -- Преобразование CLOB к верхнему регистру
    name_upper := UPPER(name);

    -- Сравнение двух CLOB
```

```

IF name = name_upper THEN
    DBMS_OUTPUT.PUT_LINE('We did not need to uppercase the name.');
```

-- Конкатенация CLOB со строками VARCHAR2

```

IF INSTR(directions, 'Mackinac Bridge') <> 0 THEN
    DBMS_OUTPUT.PUT_LINE('To get to ' || name_upper || blank_space
                          || 'you must cross the Mackinac Bridge.');
```

END IF;

END;

Результат:

To get to MUNISING FALLS you must cross the Mackinac Bridge.

Маленький фрагмент кода в этом примере содержит несколько интересных моментов:

- Столбец `falls_name` имеет тип `VARCHAR2`, однако он читается в переменную `CLOB`, что демонстрирует неявное преобразование между типами `VARCHAR2` и `CLOB`.
- Функция `SUBSTR` ограничивает выборку первыми 500 символами описания, а функция `UPPER` преобразует название водопада к верхнему регистру. Тем самым демонстрируется применение функций SQL и PL/SQL к LOB.
- Команда `IF`, сравнивающая `name` с `name_upper`, выглядит немного неестественно, но она демонстрирует возможность применения операторов отношения к LOB.
- Название, приведенное к верхнему регистру — `CLOB`, — объединяется со строковыми константами и одной строкой `VARCHAR2` (`blank_space`). Тем самым демонстрируется возможность конкатенации `CLOB`.

При использовании этой функциональности приходится учитывать многочисленные ограничения и скрытые ловушки. Например, не каждая функция, получающая значение `VARCHAR2`, примет вместо него `CLOB`; есть и исключения. В частности, функции регулярных выражений работают с семантикой SQL, а агрегатные функции — нет; не все операторы отношений поддерживаются для LOB и т. д. Все эти ограничения подробно описываются в разделе «SQL Semantics and LOBs» главы 10 руководства *SecureFiles and Large Objects Developer's Guide* для Oracle Database 11g и 12c. Для Oracle Database 10g обращайтесь к главе 9, «SQL Semantics and LOB», руководства *Application Developers Guide — Large Objects*. Если вы собираетесь использовать семантику SQL, я настоятельно рекомендую ознакомиться с соответствующим разделом руководства для вашей базы данных.



Семантика SQL для LOB действует только для внутренних типов LOB: `CLOB`, `BLOB` и `NCLOB`. Поддержка семантики SQL не относится к `BFILE`.

Семантика SQL может создавать временные объекты LOB

При применении семантики SQL к LOB необходимо учитывать один важный аспект: она часто приводит к созданию временного объекта LOB. Подумайте, как функция `UPPER` применяется к `CLOB`:

```

DECLARE
    directions CLOB;
BEGIN
    SELECT UPPER(falls_directions)
        INTO directions
        FROM waterfalls
        WHERE falls_name = 'Munising Falls';
END;
```

Так как объем данных CLOB очень велик, эти объекты хранятся на диске. База данных не может преобразовать к верхнему регистру читаемый объект CLOB, потому что это означает изменение его значения на диске — по сути это означает изменение читаемого значения. Также база данных не может внести изменения в копию CLOB в памяти, потому что значение может не поместиться в памяти, а в результате выборки программа получает только локатор, который указывает на значение на диске. Единственное, что может сделать база данных, — создать временный объект CLOB во временном пространстве. Функция UPPER копирует данные из исходного объекта CLOB во временный объект CLOB, преобразуя символы к верхнему регистру в процессе копирования. Затем команда SELECT возвращает локатор LOB, указывающий на временный, а не на исходный объект CLOB. У этого факта есть два очень важных следствия:

- Локатор, возвращаемый функцией или выражением, не может использоваться для обновления исходного объекта LOB. Переменная `directions` в моем примере не может использоваться для обновления постоянного объекта LOB, хранящегося в базе данных, потому что она в действительности указывает на временный объект LOB, возвращаемый функцией UPPER.
- Создание временного объекта LOB требует расхода дискового пространства и вычислительных ресурсов. Эта тема более подробно рассматривается в разделе «Влияние семантики SQL на быстродействие».

Чтобы получить описание местонахождения водопада в верхнем регистре, сохранив возможность обновления исходного описания, необходимо получить два локатора LOB:

```
DECLARE
    directions_upper CLOB;
    directions_persistent CLOB;
BEGIN
    SELECT UPPER(falls_directions), falls_directions
        INTO directions_upper, directions_persistent
        FROM waterfalls
        WHERE falls_name = 'Munising Falls';
END;
```

Теперь я могу обратиться к описанию в верхнем регистре через локатор в `directions_upper` и изменить исходное описание через локатор в `directions_persistent`. Выборка лишнего локатора в данном случае не отражается на быстродействии приложения — на него влияет преобразование текста к верхнему регистру и его сохранение во временном объекте CLOB. Локатор в `directions_persistent` просто читается из таблицы базы данных.

В общем случае любая символьно-строковая функция, которая обычно получает VARCHAR2 и возвращает VARCHAR2, при передаче CLOB вернет временный объект CLOB. Аналогичным образом выражения, возвращающие CLOB, почти наверняка будут возвращать временные объекты CLOB. Временные объекты CLOB и BLOB не могут использоваться для обновления данных LOB, использованных в выражении или функции.

Влияние семантики SQL на быстродействие

Используя новую семантику SQL при работе с LOB, стоит учесть ее последствия для быстродействия. Помните, что данные LOB могут занимать до 128 терабайт (4 гигабайта до Oracle Database 10g). Соответственно, при неосмотрительном использовании LOB как любого другого типа переменной или столбца могут возникнуть серьезные проблемы. Взгляните на следующий запрос, который пытается найти все водопады, путь к которым проходит через мост Макинак:

```
SELECT falls_name
    FROM waterfalls
   WHERE INSTR(UPPER(falls_directions), 'MACKINAC BRIDGE') <> 0;
```

Представьте, как должна действовать база данных Oracle при обработке этого запроса. Для каждой строки в таблице `waterfalls` она должна взять содержимое столбца `falls_directions`, преобразовать его к верхнему регистру и поместить результаты во временный объект `CLOB` (находящийся во временном табличном пространстве). Затем ко временному объекту `LOB` применяется функция `INSTR`, которая ищет строку «MACKINAC BRIDGE». В моих примерах описания были относительно короткими. Но представьте, что `falls_directions` содержит действительно большие данные `LOB`, а средний размер столбца составляет 1 Гбайт. Естественно, выделение пространства для всех временных объектов `LOB` при преобразовании описаний к верхнему регистру приведет к быстрому расходованию временного табличного пространства.

А теперь представьте, сколько времени потребуется для создания копии каждого объекта `CLOB` для его преобразования к верхнему регистру, для выделения и освобождения пространства временных объектов `CLOB` во временном пространстве и для посимвольного поиска данных в `CLOB` средним объемом в 1 Гбайт. Конечно, такие запросы навлекут на вас гнев администратора базы данных.

ORACLE TEXT И СЕМАНТИКА SQL

Если вам понадобится выполнить запросы с эффективным поиском в `CLOB`, возможно, вам поможет Oracle Text. Допустим, когда-нибудь вам потребуется написать запрос следующего вида:

```
SELECT falls_name
FROM waterfalls
WHERE INSTR(UPPER(falls_directions), 'MACKINAC BRIDGE') <> 0;
```

Если `falls_directions` является столбцом `CLOB`, эффективность такого запроса оставляет желать лучшего. Однако при использовании Oracle Text вы можете определить для этого столбца индекс Oracle Text, игнорирующий регистр символов, после чего воспользоваться предикатом `CONTAINS` для эффективной обработки запроса:

```
SELECT falls_name
FROM waterfalls
WHERE
CONTAINS(falls_directions,'mackinac bridge') > 0;
```

За дополнительной информацией о `CONTAINS` и индексах Oracle Text, игнорирующих регистр символов, обращайтесь к документации Oracle Text Application Developer's Guide.

Из-за возможных негативных последствий от применения семантики SQL к `LOB` в документации Oracle рекомендуется ограничить ее данными `LOB`, размер которых не превышает 100 Кбайт. Я не готов предложить конкретные рекомендации по размеру; рассмотрите каждую ситуацию в контексте своих обстоятельств. Всегда учитывайте возможные последствия от применения семантики SQL к `LOB`; возможно, вам стоит провести тестирование для оценки этих последствий, чтобы принять разумное решение в вашей конкретной ситуации.

Функции преобразования объектов LOB

Oracle предоставляет в распоряжение программиста несколько функций преобразования больших объектов. Перечень этих функций приведен в табл. 13.2.

Функция `TO_LOB` предназначена для одноразового перемещения данных `LONG` и `LONG RAW` в столбцы типа `CLOB` и `BLOB`, поскольку типы `LONG` и `LONG RAW` теперь считаются устаревшими. Функции `TO_CLOB` и `TO_NCLOB` предоставляют удобные средства преобразования символьных данных больших объектов и наборов символов базы данных и национальных языков.

Таблица 13.2. Функции преобразования больших объектов

| Функция | Описание |
|-----------------------------|---|
| TO_CLOB(символьные_данные) | Преобразует символьные данные (типы VARCHAR2, NVARCHAR2, CHAR, NCHAR, CLOB и NCLOB) в данные типа CLOB. При необходимости (для входных данных типа NVARCHAR2) данные преобразуются из национального набора в набор символов базы данных |
| TO_BLOB(данные_типа_raw) | Аналог функции TO_CLOB; преобразует данные RAW или LONG RAW к типу BLOB |
| TO_NCLOB(символьные_данные) | Аналог функции TO_CLOB; результат имеет тип NCLOB и представлен символами национального набора |
| TO_LOB(данные_long) | Принимает значение типа LONG или LONG RAW и преобразует эти данные в CLOB или BLOB соответственно. Функция TO_LOB может быть вызвана только из списка выборки подзапроса в команде INSERT...SELECT...FROM. |
| TO_RAW | Принимает значение типа BLOB и возвращает его как значение типа RAW |

Предопределенные объектные типы

В Oracle9i был введен ряд предопределенных объектных типов:

- **XMLType** — хранение и обработка данных в XML-формате;
- типы **URI** — хранение унифицированных идентификаторов ресурсов (в частности, HTML-адресов);
- типы **Any** — определение переменных PL/SQL, в которых могут храниться данные любых типов.

Тип XMLType

В Oracle9i появился встроенный объектный тип **XMLType** для определения столбцов и переменных PL/SQL, содержащих документы XML. Методы **XMLType** позволяют создавать экземпляры новых значений **XMLType**, извлекать фрагменты документов XML и выполнять другие операции с содержимым документов XML.

Язык XML — обширная тема, которую невозможно подробно изложить в книге. Тем не менее при работе с XML из PL/SQL необходимо знать как минимум две вещи:

- **XMLType** — встроенный объектный тип, который позволяет хранить документы XML в столбце базы данных или в переменной PL/SQL. Тип **XMLType** был введен в Oracle9i Release 1.
- **XQuery** — язык запросов для выборки и построения документов XML. Поддержка **XQuery** появилась в Oracle10g Release 2.

Кроме этих двух технологий, в работе с XML также используются технологии XPath для построения ссылок на части документа, XML Schema для описания структуры документа и т. д. Тип **XMLType** позволяет легко создать таблицу для хранения данных XML:

```
CREATE TABLE fallsXML (
    fall_id NUMBER,
    fall XMLType
);
```

В этой таблице для XML-данных определен столбец **fall** с типом **XMLType**. Чтобы записать в него информацию, необходимо вызвать статический метод **CreateXML** и передать ему данные в формате XML. Полученный объект возвращается как результат метода и помещается в столбец базы данных. Перегруженные версии метода **CreateXML** могут получать как данные **VARCHAR2**, так и данные **CLOB**.

Следующие инструкции INSERT создают три документа XML и помещают их в таблицу falls:

```
INSERT INTO fallsXML VALUES (1, XMLType.CreateXML(
  '<?xml version="1.0"?>
  <fall>
    <name>Munising Falls</name>
    <county>Alger</county>
    <state>MI</state>
    <url>
      http://michiganwaterfalls.com/munising_falls/munising_falls.html
    </url>
  </fall>' ));

INSERT INTO fallsXML VALUES (2, XMLType.CreateXML(
  '<?xml version="1.0"?>
  <fall>
    <name>Au Train Falls</name>
    <county>Alger</county>
    <state>MI</state>
    <url>
      http://michiganwaterfalls.com/autrain_falls/autrain_falls.html
    </url>
  </fall>' ));

INSERT INTO fallsXML VALUES (3, XMLType.CreateXML(
  '<?xml version="1.0"?>
  <fall>
    <name>Laughing Whitefish Falls</name>
    <county>Alger</county>
    <state>MI</state>
    <url>
      http://michiganwaterfalls.com/whitefish_falls/whitefish_falls.html
    </url>
  </fall>' ));
```

Для выборки XML-данных из таблицы используются методы объекта XMLType. Метод existsNode, вызываемый в следующем примере, проверяет существование в XML-документе заданного узла. Аналогичную проверку выполняет встроенная функция SQL EXISTSNode. Она, как и указанный метод, идентифицирует узел с помощью выражения XPath¹.

Следующие инструкции возвращают одинаковые результаты:

```
SQL> SELECT f.fall_id
2 FROM fallsxml f
3 WHERE f.fall.existsNode('/fall/url') > 0;

SQL> SELECT f.fall_id
2 FROM fallsxml f
3 WHERE EXISTSNode(f.fall, '/fall/url') > 0;
4 /

FALL_ID
-----
1
2
```

Конечно, с XML-данными можно работать и в PL/SQL. В следующем примере переменной PL/SQL типа XMLType присваивается значение из столбца fall первой добавленной нами строки таблицы. Затем я считываю в программе PL/SQL весь XML-документ,

¹ Синтаксис XPath используется для описания частей XML-документа. Кроме того, XPath позволяет задать конкретный узел или значение атрибута документа.

с которым после этого можно работать, как с любой другой информацией. В данном случае после выборки документа мы извлекаем и выводим текст из узла `/fall/url`.

```
<<demo_block>>
DECLARE
    fall XMLType;
    url VARCHAR2(100);
BEGIN
    --Выборка XML-данных
    SELECT f.fall
        INTO demo_block.fall
        FROM fallsXML f
        WHERE f.fall_id = 1;

    -- Извлечение и вывод URL
    url := fall.extract('/fall/url/text()').getStringVal;
    DBMS_OUTPUT.PUT_LINE(url);
END;
```

Обратите внимание на следующие строки:

- `SELECT f.fall INTO demo_block.fall` — имя переменной `fall` совпадает с именем столбца таблицы, поэтому в запросе SQL имя переменной уточняется именем блока PL/SQL.
- `url := fall.extract('/fall/url/text()').getStringVal;` — для получения текста URL вызываются два метода объекта XMLType:
 - `extract` — возвращает объект XMLType, содержащий заданный фрагмент исходного XML-документа (для определения требуемого фрагмента используется выражение XPath);
 - `getStringVal` — возвращает текст XML-документа.

В рассмотренном примере метод `getStringVal` вызывается для XML-документа, возвращаемого методом `extract`. Метод `extract` возвращает содержимое узла `<url>` в виде объекта XMLType, а метод `getStringVal` извлекает из него содержимое в виде текстовой строки.

Столбцы XMLType даже можно индексировать для повышения эффективности выборки XML-документов. Для создания индекса необходимо обладать привилегиями `QUERY REWRITE`. В следующем примере индекс строится по первым 80 символам имени водоппада из таблицы `falls`:

```
CREATE INDEX falls_by_name
ON fallsxml f (
    SUBSTR(
        XMLType.getStringVal(
            XMLType.extract(f.fall, '/fall/name/text()')
        ), 1, 80
    )
)
```

Обратите внимание на использование функции `SUBSTR`. Метод `getStringVal` возвращает строку, слишком длинную для индексирования, в результате чего происходит ошибка. Функция же `SUBSTR` уменьшает длину полученной строки до приемлемого размера.

Если вы решите задействовать объект XMLType в своих приложениях, за более полной и актуальной информацией обращайтесь к документации Oracle. *XML DB Developer's Guide* содержит важную, если не сказать — абсолютно необходимую информацию для разработчиков, работающих с XML. В *SQL Reference* также представлена полезная информация о XMLType и встроенных функциях SQL, поддерживающих работу с XML. В справочнике Oracle *PL/SQL Packages and Types Reference* описаны программы, методы и исключения всех предопределенных объектных типов, а также нескольких пакетов для работы с данными XML, включая `DBMS_XDB`, `DBMS_XMLSCHEMA` и `DBMS_XMLDOM`.

Типы данных URI

Семейство типов URI представлено одним основным типом данных и несколькими под-типами, обеспечивающими поддержку хранения URI в переменных PL/SQL и столбцах баз данных. Основной тип для работы с URI называется UriType; в переменной этого типа может храниться экземпляр любого из следующих подтипов:

- `HttpUriType` — специфический для URL подтип, обычно идентифицирующий веб-страницы;
- `DBUriType` — подтип для поддержки URL, представленных в виде выражений XPath;
- `XDBUriType` — подтип для поддержки URL, идентифицирующих объекты Oracle XML DB (набор XML-технологий, встроенных в базу данных Oracle).

Также Oracle предоставляет пакет `UriFactory`, автоматически генерирующий правильный тип для переданного URI.

Типы URI создаются сценарием `$ORACLE_HOME/rdbms/admin/dbmsuri.sql`. Владелец всех типов и подтипов является пользователь `SYS`.

Начиная с Oracle11g, включение сетевого доступа требует создания и настройки списков ACL (Access Control List). Это усовершенствование из области безопасности требует выполнения ряда предварительных условий до выхода в Интернет: вы должны создать сетевой список ACL, предоставить ему необходимые привилегии, а затем определить те адреса, к которым разрешает доступ список ACL.

```
BEGIN
-- Создание списка ACL
DBMS_NETWORK_ACL_ADMIN.CREATE_ACL(
    acl          => 'oreillynet-permissions.xml'
  ,description  => 'Network permissions for www.oreillynet.com'
  ,principal   => 'WEBROLE'
  ,is_grant    => TRUE
  ,privilege    => 'connect'
  ,start_date  => SYSTIMESTAMP
  ,end_date    => NULL
);
-- Назначение привилегий
DBMS_NETWORK_ACL_ADMIN.ADD_PRIVILEGE (
    acl          => 'oreillynet-permissions.xml'
  ,principal    => 'WEBROLE'
  ,is_grant     => TRUE
  ,privilege    => 'connect'
  ,start_date   => SYSTIMESTAMP
  ,end_date     => null
);
-- Определение допустимых адресов
DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL (
    acl          => 'oreillynet-permissions.xml'
  ,host         => 'www.oreillynet.com'
  ,lower_port   => 80
  ,upper_port   => 80
);
COMMIT; -- Изменения необходимо закрепить.
END;
```

Теперь можно переходить к выборке веб-страниц с использованием типа `HttpUriType`:

```
DECLARE
    WebPageURL HttpUriType;
    WebPage CLOB;
BEGIN
    -- Создание экземпляра объектного типа, идентифицирующего
    -- страницу на сайте издательства O'Reilly
    WebPageURL := HttpUriType.createUri('http://www.oreillynet.com/pub/au/344');
```

продолжение ➤

```
-- Получение сообщения по протоколу HTTP
WebPage := WebPageURL.getclob();

-- Вывод заголовка страницы
DBMS_OUTPUT.PUT_LINE(REGEXP_SUBSTR(WebPage, '<title>.*</title>'));
END;
```

В результате выполнения выводится следующий текст:

```
<title>Steven Feuerstein</title>
```

За дополнительной информацией о типах семейства `UriType` обращайтесь к главе 20 документации *XML DB Developer's Guide*.

Типы данных Any

Как упоминалось в главе 7, PL/SQL относится к числу языков со статической типизацией. Как правило, типы данных объявляются и проверяются во время компиляции. Иногда бывает не обойтись без средств динамической типизации; для таких случаев в Oracle9i Release 1 были введены типы `Any`. Они позволяют выполнять операции над данными, тип которых неизвестен до выполнения программы. При этом поддерживается механизм *интроспекции*, позволяющий определить тип значения во время выполнения и обратиться к этому значению.



Механизм интроспекции может использоваться в программах для анализа и получения информации о переменных, объявленных в программе. По сути, программа получает информацию о самой себе — отсюда и термин «интроспекция».

Типы `Any` непрозрачны, то есть вы не можете манипулировать с внутренними структурами напрямую, а должны использовать программные средства.

К семейству `Any` относятся следующие типы данных:

- `AnyData` — может содержать одиночное значение любого типа: скалярную величину, пользовательский объект, вложенную таблицу, массив `VARRAY` и т. д.
- `AnyDataSet` — может содержать набор значений, относящихся к одному типу.
- `AnyType` — описание типа, своего рода «тип без данных».

Типы `Any` включаются в исходную поставку базы данных или создаются сценарием `dbmsany.sql`, хранящимся в каталоге `$ORACLE_HOME/rdbms/admin`. Их владельцем является пользователь `SYS`.

Кроме типов `Any`, сценарий `dbmsany.sql` создает пакет `DBMS_TYPES` с определениями именованных констант (таких, как `TYPECODE_DATE`). Они могут использоваться совместно с анализирующими функциями, и в частности с `GETTYPE`, для определения типа данных, хранящихся в конкретной переменной `AnyData` или `AnyDataSet`. Конкретные числовые значения этих констант для нас несущественны — ведь константы для того и определены, чтобы программисты пользовались именами, а не значениями.

В следующем примере создаются два пользовательских типа, представляющих два географических объекта: водопады и реки. Далее блок кода PL/SQL с помощью функции `SYS.AnyType` определяет массив разнородных объектов (элементы которого могут относиться к разным типам данным).

Сначала создаются два объектных типа:

```
/* Файл в Сети: ch13_anydata.sql */
TYPE waterfall AS OBJECT (
    name VARCHAR2(30),
    height NUMBER
);
```

```
TYPE river AS OBJECT (
  name VARCHAR2(30),
  length NUMBER
)
```

Затем выполняется следующий блок PL/SQL:

```
DECLARE
  TYPE feature_array IS VARRAY(2) OF SYS.AnyData;
  features feature_array;
  wf waterfall;
  rv river;
  ret_val NUMBER;
BEGIN
  -- Создание массива, элементы которого
  -- относятся к разным объектным типам
  features := feature_array(
    AnyData.ConvertObject(
      waterfall('Grand Sable Falls',30)),
    AnyData.ConvertObject(
      river('Manistique River', 85.40))
  );

  -- Вывод данных
  FOR x IN 1..features.COUNT LOOP
    -- Выполнение кода, соответствующего текущему
    -- объектному типу. ВНИМАНИЕ! GetTypeName возвращает
    -- SchemaName.TypeName, поэтому PLUSER следует
    -- заменить именем используемой схемы.
    CASE features(x).GetTypeName
    WHEN 'PLUSER.WATERFALL' THEN
      ret_val := features(x).GetObject(wf);
      DBMS_OUTPUT.PUT_LINE('Waterfall: '
        || wf.name || ', Height = ' || wf.height || ' feet.');
```

Результат выполнения кода будет таким:

```
Waterfall: Grand Sable Falls, Height = 30 feet.
River: Manistique River, Length = 85.4 miles.
```

Давайте разберемся, как работает этот код. Необходимые для его работы объекты хранятся в массиве VARRAY, который инициализируется следующим образом:

```
features := feature_array(
  AnyData.ConvertObject(
    waterfall('Grand Sable Falls',30)),
  AnyData.ConvertObject(
    river('Manistique River', 85.40))
);
```

Рассмотрим в общих чертах структуру этого кода:

- waterfall('Grand Sable Falls',30)

Вызов конструктора типа `waterfall` для создания объекта типа `waterfall`.

- AnyData.ConvertObject(

Преобразование объекта `waterfall` в экземпляр `SYS.AnyData`, который после этого можно будет записать в массив объектов `SYS.AnyData`.

○ feature_array (

Вызов конструктора массива. Каждый аргумент этой функции имеет тип `AnyData`.

В данном случае массив состоит из двух передаваемых аргументов.

Массивы `VARRAY` рассматривались в главе 12, а объектные типы будут более подробно описаны в главе 26.

Следующий важный фрагмент кода — цикл `FOR`, в котором последовательно анализируются объекты массива `features`. Вызов `features(x).GetTypeName` возвращает полное имя текущего объекта. Для пользовательских объектов перед именем типа ставится имя схемы пользователя, создавшего объект. Это имя схемы включается в условие `WHEN`:

```
WHEN 'PLUSER.WATERFALL' THEN
```

Если вы захотите выполнить этот пример в своей системе, не забудьте заменить используемую схему (`PLUSER`) именем своей схемы. При создании типов, которые должны использоваться при интроспекции, тщательно продумайте проблему владельца типа — возможно, вам придется статически включить его в код.



При использовании встроенных типов данных, таких как `NUMBER`, `DATE` и `VARCHAR2`, функция `GetTypeName` возвращает просто имя типа. Имя схемы указывается только для типов, определяемых пользователем (то есть созданных конструкцией `CREATE TYPE`).

Определив тип данных, мы извлекаем объект из массива:

```
ret_val := features(x).GetObject(wf);
```

В данном примере возвращаемое функцией значение игнорируется. В общем случае результатом работы функции может быть одно из двух значений:

- `DBMS_TYPES.SUCCESS` — значение, свидетельствующее о том, что переменная типа `Any` содержит данные определенного типа (в нашем случае объект).
- `DBMS_TYPES.NO_DATA` — значение, указывающее, что в переменной типа `AnyData` не оказалось никаких данных.

Когда переменной будет присвоен экземпляр объекта, мы можем относительно легко написать оператор `DBMS_OUTPUT` для объекта данного типа. Например, информация о водопаде выводится так:

```
DBMS_OUTPUT.PUT_LINE('Waterfall: '
|| wf.name || ', Height = ' || wf.height || ' feet.');
```

За дополнительной информацией о типах данных `Any` обращайтесь к главе 26, где они рассматриваются с позиций объектно-ориентированного программирования. Также желательно ознакомиться с документами Oracle *PL/SQL Packages and Types Reference* и *Object-Relational Developer's Guide*. Наконец, ознакомьтесь со сценариями `anynums.pkg` и `anynums.tst` на сайте книги.



С точки зрения объектно-ориентированного программирования существуют более эффективные способы работы с объектными типами данных. Однако на практике не всегда стоит тратить время на достижение идеала, и рассмотренный пример достаточно хорошо демонстрирует возможности предопределенного объектного типа `SYS.AnyData`.

IV SQL и PL/SQL

Эта часть книги посвящена одному из центральных аспектов кода PL/SQL — взаимодействию с базами данных средствами языка SQL. В главах 14–16 рассказывается о том, как определять транзакции для обновления, вставки и удаления информации в базах данных; как запрашивать из базы данных информацию для обработки в программах PL/SQL; как выполнять SQL-инструкции с использованием встроенного динамического SQL (NDS).

14

DML и управление транзакциями

Язык PL/SQL плотно интегрирован с базой данных Oracle. Из кода PL/SQL можно выполнять любые команды DML (Data Manipulation Language), в том числе `INSERT`, `UPDATE`, `DELETE` и `MERGE` и, конечно же, запросы на выборку.



Команды DDL (Data Definition Language) выполняются только в режиме динамического SQL (см. главу 16).

Несколько SQL-команд можно сгруппировать на логическом уровне в одну *транзакцию*, чтобы их результаты либо все вместе сохранялись (*закрепление*), либо все вместе отменялись (*откат*). В этой главе рассматриваются SQL-команды, используемые в PL/SQL для управления транзакциями.

Чтобы оценить важность транзакций в Oracle, необходимо хорошо понимать их основные свойства:

- **Атомарность.** Вносимые в ходе транзакции изменения состояния имеют атомарный характер: либо выполняются все изменения сразу, либо не выполняется ни одно из них.
- **Согласованность.** Транзакция корректно изменяет состояние базы данных. Действия, выполняемые как единое целое, не нарушают ограничений целостности, связанных с данным состоянием.
- **Изолированность.** Возможно параллельное выполнение множества транзакций, но с точки зрения каждой конкретной транзакции остальные кажутся выполняемыми до или после нее.
- **Устойчивость.** После успешного завершения транзакции измененные данные закрепляются в базе данных и становятся устойчивыми к последующим сбоям.

Начатая транзакция либо фиксируется командой `COMMIT`, либо отменяется командой `ROLLBACK`. В любом случае будут освобождены заблокированные транзакцией ресурсы (команда `ROLLBACK TO` может снять только часть блокировок). Затем сеанс, как правило, начинает новую транзакцию. По умолчанию в PL/SQL неявно определяется одна транзакция на весь сеанс, и все выполняемые в ходе этого сеанса изменения данных являются частью транзакции. Однако применение технологии *автономных транзакций* позволяет определять вложенные транзакции, выполняемые внутри главной транзакции уровня сеанса.

DML в PL/SQL

Из блока кода PL/SQL можно выполнять DML-команды (**INSERT**, **UPDATE**, **DELETE** и **MERGE**), оперирующие любыми доступными таблицами и представлениями.



При использовании модели разрешений создателя права доступа к этим структурам определяются во время компиляции, если вы используете модель прав определяющей стороны. Если же используется модель разрешений вызывающей стороны с конструкцией **AUTHID CURRENT_USER**, то права доступа определяются во время выполнения программы. Подробнее об этом рассказывается в главе 24.

Краткое введение в DML

Полное описание всех возможностей DML в языке Oracle SQL выходит за рамки книги, поэтому мы ограничимся кратким обзором базового синтаксиса, а затем изучим специальные возможности PL/SQL, относящиеся к DML, включая:

- примеры команд DML;
- атрибуты курсоров команд DML;
- синтаксис PL/SQL, относящийся к DML, например конструкция **RETURNING**.

За более подробной информацией обращайтесь к документации Oracle и другим описаниям SQL.



Формально команда **SELECT** считается командой DML. Однако разработчики под термином «DML» почти всегда понимают команды, изменяющие содержимое таблицы базы данных (то есть не связанные с простым чтением данных).

В языке SQL определены четыре команды DML:

- **INSERT** — вставляет в таблицу одну или несколько новых строк.
- **UPDATE** — обновляет в одной или нескольких существующих строках таблицы значения одного или нескольких столбцов.
- **DELETE** — удаляет из таблицы одну или несколько строк.
- **MERGE** — если строка с заданными значениями столбцов уже существует, выполняет обновление. В противном случае выполняется вставка.

Команда INSERT

Существует две базовые разновидности команды **INSERT**:

- Вставка одной строки с явно заданным списком значений:

```
INSERT INTO таблица [(столбец_1, столбец_2, ..., столбец_n)]  
VALUES (значение_1, значение_2, ..., значение_n);
```

- Вставка в таблицу одной или нескольких строк, определяемых командой **SELECT**, которая извлекает данные из других таблиц:

```
INSERT INTO таблица [(столбец_1, столбец_2, ..., столбец_n)]  
SELECT ...;
```

Рассмотрим несколько примеров команд **INSERT** в блоке PL/SQL. Начнем со вставки новой строки в таблицу **books**. Обратите внимание: если в секции **VALUES** заданы значения всех столбцов, то список столбцов можно опустить:

```
BEGIN
  INSERT INTO books
    VALUES ('1-56592-335-9',
      'Oracle PL/SQL Programming',
      'Reference for PL/SQL developers,' ||
      'including examples and best practice ' ||
      'recommendations.',
      'Feuerstein,Steven, with Bill Pribyl',
      TO_DATE ('01-SEP-1997','DD-MON-YYYY'),
      987);
END;
```

Можно также задать список имен столбцов, а их значения указать в виде переменных, а не литералов:

```
DECLARE
  l_isbn books.isbn%TYPE := '1-56592-335-9';
  ... другие объявления локальных переменных
BEGIN
  INSERT INTO books (
    book_id, isbn, title, summary, author,
    date_published, page_count)
  VALUES (
    book_id_sequence.NEXTVAL, l_isbn, l_title, l_summary, l_author,
    l_date_published, l_page_count);
```

ВСТРОЕННАЯ ПОДДЕРЖКА ПОСЛЕДОВАТЕЛЬНОСТЕЙ В ORACLE11G

До выхода Oracle11 программисту, желавшему получить следующее значение из последовательности, приходилось вызывать функцию NEXTVAL в команде SQL. Это можно было сделать прямо в команде INSERT, которой требовалось значение:

```
INSERT INTO table_name VALUES (sequence_name.NEXTVAL, ...);
```

или в команде SELECT из старой доброй таблицы dual:

```
SELECT sequence_name.NEXTVAL INTO l_primary_key FROM SYS.dual;
```

Начиная с Oracle11g, следующее (и текущее) значение можно получить при помощи оператора присваивания — например:

```
l_primary_key := sequence_name.NEXTVAL;
```

Команда UPDATE

Команда UPDATE обновляет один или несколько столбцов или одну или несколько строк таблицы. Ее синтаксис выглядит так:

```
UPDATE таблица
  SET столбец_1 = значение_1
    [, столбец_2 = значение_2, ... столбец_N = значение_N]
[WHERE условие];
```

Предложение WHERE не обязательно; если оно не задано, обновляются все строки таблицы. Несколько примеров команды UPDATE:

- Перевод названий книг таблицы books в верхний регистр:


```
UPDATE books SET title = UPPER (title);
```
- Выполнение процедуры, удаляющей компонент времени из даты издания книг, которые были написаны заданным автором, и переводящей названия этих книг в символы верхнего регистра. Как видно из примера, команду UPDATE можно выполнять как саму по себе, так и в блоке PL/SQL:


```

PROCEDURE remove_time (author_in IN VARCHAR2)
IS
BEGIN
    UPDATE books
        SET title = UPPER (title),
            date_published = TRUNC (date_published)
    WHERE author LIKE author_in;
END;

```

Команда DELETE

Команда DELETE удаляет одну, несколько или все строки таблицы. Базовый синтаксис:

```

DELETE FROM таблица
[WHERE условие];

```

Условие WHERE не обязательно; если оно не задано, удаляются все строки таблицы. Несколько примеров команды DELETE:

- Удаление всей информации из таблицы books:

```
DELETE FROM books;
```

- Удаление из таблицы books всей информации о книгах, изданных до определенной даты, с возвратом их общего количества:

```

PROCEDURE remove_books (
    date_in          IN      DATE,
    removal_count_out OUT    PLS_INTEGER)
IS
BEGIN
    DELETE FROM books WHERE date_published < date_in;
    removal_count_out := SQL%ROWCOUNT;
END;

```

Конечно, все эти команды DML в реальных приложениях обычно бывают гораздо сложнее. Например, команда может обновлять сразу несколько столбцов с данными, сгенерированными вложенным запросом. Начиная с Oracle9i, имя таблицы можно заменить *табличной функцией*, возвращающей результирующий набор строк, с которыми работает команда DML (см. главу 17).

Команда MERGE

В команде MERGE задается условие проверки, а также два действия для его выполнения (MATCHED) или невыполнения (NOT MATCHED). Пример:

```

PROCEDURE time_use_merge (dept_in IN employees.department_id%TYPE)
)
IS
BEGIN
    MERGE INTO bonuses d
        USING (SELECT employee_id, salary, department_id
                FROM employees
                WHERE department_id = dept_in) s
        ON (d.employee_id = s.employee_id)
    WHEN MATCHED
    THEN
        UPDATE SET d.bonus = d.bonus + s.salary * .01
    WHEN NOT MATCHED
    THEN
        INSERT (d.employee_id, d.bonus)
            VALUES (s.employee_id, s.salary * 0.2
END;

```

Синтаксис команды MERGE в значительной мере привязан к SQL, и в книге он подробно не рассматривается. Более подробный пример содержится в файле `merge.sql` на сайте книги.

Атрибуты курсора для операций DML

Для доступа к информации о последней операции, выполненной командой SQL, Oracle предоставляет несколько атрибутов курсоров, неявно открываемых для этой операции. Атрибуты неявных курсоров возвращают информацию о выполнении команд INSERT, UPDATE, DELETE, MERGE или SELECT INTO. Атрибуты курсора для команды SELECT INTO рассматриваются в главе 15. В этом разделе речь пойдет об использовании атрибутов SQL% для команд DML.

Следует помнить, что значения атрибутов неявного курсора всегда относятся к последней выполненной команде SQL, независимо от того, в каком блоке выполнялся неявный курсор. До открытия первого SQL-курсора сеанса значения всех неявных атрибутов равны NULL. (Исключение составляет атрибут %ISOPEN, который возвращает FALSE.)

Значения, возвращаемые атрибутами неявных курсоров, описаны в табл. 14.1.

Таблица 14.1. Атрибуты неявных курсоров для команд DML

| Имя | Описание |
|--------------|---|
| SQL%FOUND | Возвращает TRUE, если хотя бы одна строка была успешно модифицирована (создана, изменена или удалена) |
| SQL%NOTFOUND | Возвращает TRUE, если команда DML не модифицировала ни одной строки |
| SQL%ROWCOUNT | Возвращает количество строк, модифицированных командой DML |
| SQL%ISOPEN | Для неявных курсоров всегда возвращает FALSE, поскольку Oracle закрывает и открывает их автоматически |

Давайте посмотрим, как эти атрибуты используются.

- С помощью атрибута SQL%FOUND можно определить, обработала ли команда DML хотя бы одну строку. Допустим, автор издает свои произведения под разными именами, а записи с информацией обо всех книгах данного автора необходимо время от времени обновлять. Эту задачу выполняет процедура, обновляющая данные столбца author и возвращающая логический признак, который сообщает, было ли произведено хотя бы одно обновление:

```
PROCEDURE change_author_name (
    old_name_in      IN      books.author%TYPE,
    new_name_in      IN      books.author%TYPE,
    changes_made_out OUT      BOOLEAN)
IS
BEGIN
    UPDATE books
        SET author = new_name_in
        WHERE author = old_name_in;

    changes_made_out := SQL%FOUND;
END;
```

- Атрибут SQL%ROWCOUNT позволяет выяснить, сколько строк обработала команда DML. Новая версия приведенной выше процедуры возвращает более полную информацию:

```
PROCEDURE change_author_name (
    old_name_in      IN      books.author%TYPE,
    new_name_in      IN      books.author%TYPE,
    rename_count_out OUT      PLS_INTEGER)
IS
BEGIN
    UPDATE books
        SET author = new_name_in
        WHERE author = old_name_in;
    rename_count_out := SQL%ROWCOUNT;
END;
```

Секция RETURNING в командах DML

Допустим, вы выполнили команду UPDATE или DELETE и хотите получить ее результаты для дальнейшей обработки. Вместо того чтобы выполнять отдельный запрос, можно включить в команду условие RETURNING, с которым нужная информация будет записана непосредственно в переменные программы. Это позволяет сократить сетевой трафик и затраты ресурсов сервера, а также свести к минимуму количество курсоров, открываемых и используемых приложением.

Рассмотрим несколько примеров, демонстрирующих эту возможность.

В следующем блоке секция RETURNING записывает в переменную новый оклад работника, вычисляемый командой UPDATE:

```
DECLARE
    myname employees.last_name%TYPE;
    mysal employees.salary%TYPE;
BEGIN
    FOR rec IN (SELECT * FROM employees)
    LOOP
        UPDATE employees
            SET salary = salary * 1.5
            WHERE employee_id = rec.employee_id
        RETURNING salary, last_name INTO mysal, myname;
        DBMS_OUTPUT.PUT_LINE ('Новый оклад ' ||
            myname || ' = ' || mysal);
    END LOOP;
END;
```

Допустим, команда UPDATE изменяет более одной строки. В этом случае можно не просто сохранить возвращаемые значения в переменных, а записать их как элементы коллекции при помощи синтаксиса BULK COLLECT. Этот прием продемонстрирован на примере команды FORALL:

```
DECLARE
    names name_varray;
    new_salaries number_varray;
BEGIN
    populate_arrays (names, new_salaries);
    FORALL indx IN names.FIRST .. names.LAST
        UPDATE compensation
            SET salary = new_salaries (indx)
            WHERE last_name = names (indx)
        RETURNING salary BULK COLLECT INTO new_salaries;
    ...
END;
```

Команда FORALL более подробно описана в главе 21.

DML и обработка исключений

Если в блоке PL/SQL инициируется исключение, Oracle *не выполняет* откат изменений, внесенных командами DML этого блока. Логическими транзакциями приложения должен управлять программист, который и определяет, какие действия следует выполнять в этом случае. Рассмотрим следующую процедуру:

```
PROCEDURE empty_library (
    pre_empty_count OUT PLS_INTEGER)
IS
BEGIN
```

/* Реализация tabcount содержится в файле ch14_code.sql */

продолжение ➞

```

pre_empty_count := tabcount ('books');
DELETE FROM books;
RAISE NO_DATA_FOUND;
END;
```

Обратите внимание: перед инициированием исключения задается значение параметра OUT. Давайте запустим анонимный блок, вызывающий эту процедуру, и проанализируем результаты:

```

DECLARE
    table_count    NUMBER := -1;
BEGIN
    INSERT INTO books VALUES (...);
    empty_library (table_count);
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line (tabcount ('books'));
        DBMS_OUTPUT.put_line (table_count);
END;
```

Код выводит следующие значения:

```

0
-1
```

Как видите, исключение было инициировано, но строки из таблицы книг при этом остались удаленными; дело в том, что Oracle не выполняет автоматического отката изменений. С другой стороны, переменная `table_count` сохранила исходное значение.

Таким образом, в программах, выполняющих операции DML, вы сами отвечаете за откат транзакции — а вернее, решаете, *хотите* ли вы выполнить откат. Принимая решение, примите во внимание следующие соображения:

- Если для блока выполняется автономная транзакция (см. далее в этой главе), в обработчике исключения *необходимо* произвести ее откат или закрепление (чаще откат).
- Для определения области отката используются *точки сохранения*. Можно произвести откат транзакции до конкретной точки сохранения, тем самым оставив часть изменений, внесенных в течение сеанса. Точки сохранения также описаны далее в этой главе.

Если исключение передается за пределы «самого внешнего» блока (то есть остается необработанным), то в среде выполнения PL/SQL, и в частности в SQL*Plus, автоматически осуществляется откат транзакции, и все изменения отменяются.

DML и записи

В командах INSERT и DELETE можно использовать записи PL/SQL. Пример:

```

PROCEDURE set_book_info (book_in IN books%ROWTYPE)
IS
BEGIN
    INSERT INTO books VALUES book_in;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX
    THEN
        UPDATE books SET ROW = book_in
        WHERE isbn = book_in.isbn;
END;
```

Это важное нововведение облегчает работу программиста по сравнению с работой на уровне отдельных переменных или полей записи. Во-первых, код становится более компактным — с уровня отдельных значений вы поднимаетесь на уровень записей. Нет

необходимости объявлять отдельные переменные или разбивать запись на поля при передаче данных команде DML. Во-вторых, повышается надежность кода — если вы работаете с записями типа %ROWTYPE и обходитесь без явных манипуляций с полями, то в случае модификации базовых таблиц и представлений вам придется вносить значительные изменения в программный код.

В разделе «Ограничения, касающиеся операций вставки и обновления» приведен список ограничений на использование записей в командах DML. Но сначала мы посмотрим, как использовать DML на основе записей в командах INSERT и UPDATE.

Вставка на основе записей

В командах INSERT записи можно использовать как для добавления единственной строки, так и для пакетной вставки (с использованием команды FORALL). Также возможно создание записей с помощью объявления %ROWTYPE на основе таблицы, в которую производится вставка, или явного объявления командой TYPE на основе типа данных, совместимого со структурой таблицы.

Приведем несколько примеров.

- Вставка в таблицу книг данных из записи с объявлением %ROWTYPE:

```
DECLARE
    my_book books%ROWTYPE;
BEGIN
    my_book.isbn := '1-56592-335-9';
    my_book.title := 'ORACLE PL/SQL PROGRAMMING';
    my_book.summary := 'General user guide and reference';
    my_book.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
    my_book.page_count := 1000;

    INSERT INTO books VALUES my_book;
END;
```

Обратите внимание: имя записи не заключается в скобки. Если мы используем запись вида

```
INSERT INTO books VALUES (my_book);
```

Oracle выдаст сообщение об ошибке.

Также можно выполнить вставку данных, взятых из записи, тип которой определен программистом, но этот тип должен быть полностью совместим с определением %ROWTYPE. Другими словами, нельзя вставить в таблицу запись, содержащую подмножество столбцов таблицы.

- Вставка с помощью команды FORALL — этим способом в таблицу вставляются коллекции записей. За дополнительной информацией о FORALL обращайтесь к главе 21.

Обновление на основе записей

Также существует возможность обновления целой строки таблицы по данным записи PL/SQL. В следующем примере для обновления строки таблицы books используется запись, созданная со спецификацией %ROWTYPE. Обратите внимание на ключевое слово ROW, которое указывает, что вся строка обновляется данными из записи:

```
/* Файл в Сети: record_updates.sql */
DECLARE
    my_book books%ROWTYPE;
BEGIN
    my_book.isbn := '1-56592-335-9';
    my_book.title := 'ORACLE PL/SQL PROGRAMMING';
    my_book.summary := 'General user guide and reference';
```

продолжение ➤

```
my_book.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
my_book.page_count := 1000;
```

```
UPDATE books
  SET ROW = my_book
  WHERE isbn = my_book.isbn;
END;
```

Существует несколько ограничений, касающихся обновления строк на основе записей.

- При использовании ключевого слова ROW должна обновляться вся строка. Возможность обновления подмножества столбцов пока отсутствует, но не исключено, что она появится в следующих версиях Oracle. Для любых полей, значения которых остались равными NULL, соответствующему столбцу будет присвоено значение NULL.
- Обновление не может выполняться с использованием вложенного запроса.

Использование записей с условием RETURNING

В команду DML может включаться секция RETURNING, возвращающая значения столбцов (и основанных на них выражений) из обработанных строк. Возвращаемые данные могут помещаться в запись и даже в коллекцию записей:

```
/* Файл в Сети: record_updates.sql */
DECLARE
  my_book_new_info books%ROWTYPE;
  my_book_return_info books%ROWTYPE;
BEGIN
  my_book_new_info.isbn := '1-56592-335-9';
  my_book_new_info.title := 'ORACLE PL/SQL PROGRAMMING';
  my_book_new_info.summary := 'General user guide and reference';
  my_book_new_info.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
  my_book_new_info.page_count := 1000;
  UPDATE books
    SET ROW = my_book_new_info
    WHERE isbn = my_book_new_info.isbn
    RETURNING isbn, title, summary, author, date_published, page_count
    INTO my_book_return_info;
END;
```

Заметьте, что в предложении RETURNING перечисляются все столбцы таблицы. К сожалению, Oracle пока не поддерживает синтаксис с символом *.

Ограничения, касающиеся операций вставки и обновления

Если вы захотите освоить операции вставки и обновления с использованием записей, имейте в виду, что на их применение существуют определенные ограничения.

- Переменная типа записи может использоваться либо (1) в правой части секции SET команды UPDATE; (2) в предложении VALUES команды INSERT; (3) в подразделе INTO секции RETURNING.
- Ключевое слово ROW используется только в левой части приложения SET. В этом случае других предложений SET быть не может (то есть нельзя задать предложение SET со строкой, а затем предложение SET с отдельным столбцом).
- При вставке записи не следует задавать значения отдельных столбцов.
- Нельзя указывать в команде INSERT или UPDATE запись, содержащую вложенную запись (или функцию, возвращающую вложенную запись).
- Записи не могут использоваться в динамически выполняемых командах DML (EXECUTE IMMEDIATE). Это потребовало бы от Oracle поддержки динамической привязки типа записи PL/SQL с командой SQL. Oracle же поддерживает динамическую привязку только для типов данных SQL.

Управление транзакциями

Oracle поддерживает очень мощную и надежную модель транзакций. Код приложения определяет логическую последовательность выполняемых операций, результаты которой должны быть либо сохранены командой **COMMIT**, либо отменены командой **ROLLBACK**. Транзакция начинается неявно с первой команды **SQL**, выполняемой после команды **COMMIT** или **ROLLBACK** (или с начала сеанса) или же после команды **ROLLBACK TO SAVEPOINT**.

Для управления транзакциями **PL/SQL** предоставляет набор команд:

- **COMMIT** — сохраняет (фиксирует) все изменения, внесенные со времени выполнения последней команды **COMMIT** или **ROLLBACK**, и снимает все блокировки.
- **ROLLBACK** — отменяет (откатывает) все изменения, внесенные со времени выполнения последней команды **COMMIT** или **ROLLBACK**, и снимает все блокировки.
- **ROLLBACK TO SAVEPOINT** — отменяет все изменения со времени установки последней точки сохранения и снимает все блокировки, установленные в этой части кода.
- **SAVEPOINT** — устанавливает точку сохранения, после чего становится возможным частичный откат транзакции.
- **SET TRANSACTION** — позволяет начать сеанс чтения или чтения-записи, установить уровень изоляции или связать текущую транзакцию с заданным сегментом отката.
- **LOCK TABLE** — позволяет заблокировать всю таблицу в указанном режиме. (По умолчанию к таблице обычно применяется блокировка на уровне строк.)

Эти команды более подробно рассматриваются в следующих разделах.

Команда **COMMIT**

Фиксирует все изменения, внесенные в базу данных в ходе сеанса текущей транзакцией. После выполнения этой команды изменения становятся видимыми для других сеансов или пользователей. Синтаксис этой команды:

```
COMMIT [WORK] [COMMENT текст];
```

Ключевое слово **WORK** не обязательно — оно только упрощает чтение кода.

Ключевое слово **COMMENT** также не является обязательным; оно используется для задания комментария, который будет связан с текущей транзакцией. Текстом комментария должен быть заключенный в одинарные кавычки литерал длиной до 50 символов. Обычно комментарии задаются для распределенных транзакций с целью облегчения их анализа и разрешения сомнительных транзакций в среде с двухфазовой фиксацией. Они хранятся в словаре данных вместе с идентификаторами транзакций.

Обратите внимание: команда **COMMIT** снимает все блокировки таблиц, установленные во время текущего сеанса (например, для команды **SELECT FOR UPDATE**). Кроме того, она удаляет все точки сохранения, установленные после выполнения последней команды **COMMIT** или **ROLLBACK**.

После того как изменения будут закреплены, их откат становится невозможным.

Все команды в следующем фрагменте являются допустимыми применениями **COMMIT**:

```
COMMIT;  
COMMIT WORK;  
COMMIT COMMENT 'maintaining account balance'.
```

Команда **ROLLBACK**

Команда **ROLLBACK** отменяет (полностью или частично) изменения, внесенные в базу данных в текущей транзакции. Для чего это может потребоваться? Например, для исправления ошибок: **DELETE FROM orders**;

«Нет, Нет! Я хотел удалить только те заказы, которые были сделаны до мая 2005 года!» Нет проблем — достаточно выполнить команду `ROLLBACK`. Что касается программирования приложений, в случае возникновения проблем откат позволяет вернуться к исходному состоянию.

Синтаксис команды `ROLLBACK`:

```
ROLLBACK [WORK] [TO [SAVEPOINT] имя_точки_сохранения];
```

Существует две основные разновидности `ROLLBACK`: без параметров и с секцией `TO`, указывающей, до какой точки сохранения следует произвести откат. Первая отменяет все изменения, выполненные в ходе текущей транзакции, а вторая отменяет все изменения и снимает все блокировки, установленные после заданной точки сохранения. (О том, как установить в приложении точку сохранения, рассказано в следующем разделе.)

Имя_точки_сохранения представляет собой необъявленный идентификатор Oracle. Это не может быть литерал (заклученный в кавычки) или имя переменной.

Все команды `ROLLBACK` в следующем фрагменте действительны:

```
ROLLBACK;  
ROLLBACK WORK;  
ROLLBACK TO begin_cleanup;
```

При откате до заданной точки сохранения все установленные после нее точки стираются, но данная точка остается. Это означает, что можно возобновить с нее транзакцию и при необходимости снова вернуться к этой же точке сохранения.

Непосредственно перед выполнением команды `INSERT`, `UPDATE`, `MERGE` или `DELETE` PL/SQL автоматически устанавливает неявную точку сохранения, и если команда завершается ошибкой, выполняется автоматический откат до этой точки. Если в дальнейшем в ходе выполнения команды DML происходит сбой, выполняется автоматический откат до этой точки. Подобным образом отменяется только последняя команда DML.

Команда **SAVEPOINT**

Устанавливает в транзакции именованный маркер, позволяющий в случае необходимости выполнить откат до отмеченной точки сохранения. При таком откате отменяются все изменения и удаляются все блокировки после этой точки, но сохраняются изменения и блокировки, предшествовавшие ей. Синтаксис команды `SAVEPOINT`:

```
SAVEPOINT имя_точки_сохранения;
```

Здесь *имя_точки_сохранения* — необъявленный идентификатор. Он должен соответствовать общим правилам формирования идентификаторов Oracle (до 30 символов, начинается с буквы, состоит из букв, цифр и символов `#`, `$` и `_`), но объявлять его не нужно (да и невозможно).

Область действия точки сохранения не ограничивается блоком PL/SQL, в котором она установлена. Если в ходе транзакции имя точки сохранения используется повторно, эта точка просто «перемещается» в новую позицию, причем независимо от процедуры, функции или анонимного блока, в котором выполняется команда `SAVEPOINT`. Если точка сохранения устанавливается в рекурсивной программе, на самом деле на каждом уровне рекурсии она задается заново, но откат может быть возможен только к одной точке — той, что установлена последней.

Команда **SET TRANSACTION**

Команда `SET TRANSACTION` позволяет начать сеанс чтения или чтения-записи, установить уровень изоляции или связать текущую транзакцию с заданным сегментом отката. Эта

команда должна быть первой командой SQL транзакции и дважды использоваться в ходе одной транзакции не может. У нее имеются четыре разновидности.

- **SET TRANSACTION READ ONLY** — определяет текущую транзакцию доступной «только для чтения». В транзакциях этого типа всем запросам доступны лишь те изменения, которые были зафиксированы до начала транзакции. Они применяются, в частности, в медленно формируемых отчетах со множеством запросов, благодаря чему в них часто используются строго согласованные данные.
- **SET TRANSACTION READ WRITE** — определяет текущую транзакцию как операцию чтения и записи данных в таблицу.
- **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE | READ COMMITTED** — определяет способ выполнения транзакции, модифицирующей базу данных. С ее помощью можно задать один из двух уровней изоляции транзакции: **SERIALIZABLE** или **READ COMMITTED**. В первом случае команде DML, пытающейся модифицировать таблицу, которая уже изменена незафиксированной транзакцией, будет отказано в этой операции. Для выполнения этой команды в инициализационном параметре **COMPATIBLE** базы данных должна быть задана версия 7.3.0 и выше.

При установке уровня **READ COMMITTED** команда DML, которой требуется доступ к строке, заблокированной другой транзакцией, будет ждать снятия этой блокировки.

- **SET TRANSACTION USE ROLLBACK SEGMENT *имя_сегмента*** — назначает текущей транзакции заданный сегмент отката и определяет ей доступ «только для чтения». Не может использоваться совместно с командой **SET TRANSACTION READ ONLY**.



Механизм сегментов отката считается устаревшим; вместо него следует использовать средства автоматического управления отменой, введенные в Oracle9i.

Команда LOCK TABLE

Команда блокирует всю таблицу базы данных в указанном режиме. Блокировка запрещает или разрешает модификацию данных таблицы со стороны других транзакций на то время, пока вы с ней работаете. Синтаксис команды **LOCK TABLE**:

```
LOCK TABLE список_таблиц IN режим_блокировки MODE [NOWAIT];
```

Здесь *список_таблиц* — список из одной или нескольких таблиц (локальных таблиц/представлений или доступных через удаленное подключение), а *режим_блокировки* — один из шести режимов: **ROW SHARE**, **ROW EXCLUSIVE**, **SHARE UPDATE**, **SHARE**, **SHARE ROW EXCLUSIVE**, **EXCLUSIVE**.

Если команда содержит ключевое слово **NOWAIT**, база данных не ждет снятия блокировки в том случае, если нужная таблица заблокирована другим пользователем, и выдает сообщение об ошибке. Если ключевое слово **NOWAIT** не указано, Oracle ждет освобождения таблицы в течение неограниченно долгого времени. Блокировка таблицы не мешает другим пользователям считывать из нее данные.

Примеры допустимых команд **LOCK TABLE**:

```
LOCK TABLE emp IN ROW EXCLUSIVE MODE;  
LOCK TABLE emp, dept IN SHARE MODE NOWAIT;  
LOCK TABLE scott.emp@new_york IN SHARE UPDATE MODE;
```



Там, где это возможно, используйте стандартные средства блокировки Oracle. Команду **LOCK TABLE** в приложениях следует использовать только в крайних случаях и с величайшей осторожностью.

Автономные транзакции

Определяя блок PL/SQL как *автономную транзакцию*, вы тем самым изолируете выполняемые в нем команды DML от контекста транзакции вызывающего кода. Этот блок определяется как независимая транзакция, начатая другой транзакцией, которая является *главной*.

В блоке автономной транзакции главная транзакция приостанавливается. Вы выполняете SQL-операции, затем производите их фиксацию или откат и возобновляете главную транзакцию (рис. 14.1).

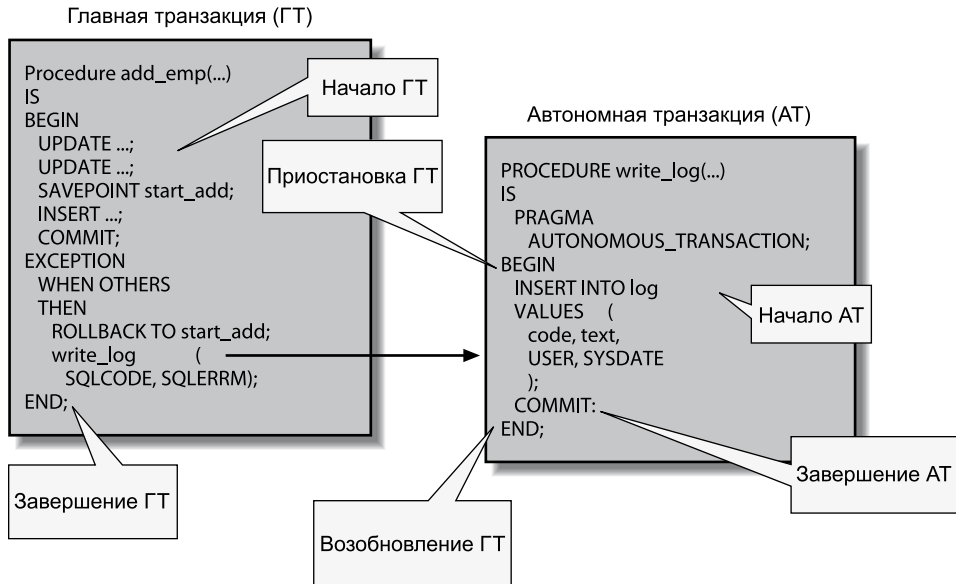


Рис. 14.1. Последовательность выполнения главной и вложенной (автономной) транзакций

Определение автономной транзакции

Определить блок PL/SQL как автономную транзакцию очень просто. Для этого достаточно включить в раздел объявлений следующую директиву:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

Директива приказывает компилятору PL/SQL назначить блок PL/SQL как автономный (независимый). В контексте автономных транзакций автономным может быть объявлен любой из следующих блоков:

- Анонимный блок PL/SQL верхнего уровня (не вложенный!).
- Функция или процедура, определенная в пакете или как отдельная программа.
- Метод (функция или процедура) объектного типа.
- Триггер базы данных.

Директива `AUTONOMOUS_TRANSACTION` может располагаться в любом месте раздела объявлений блока PL/SQL. Однако лучше всего разместить ее перед определениями структур данных, чтобы любому программисту сразу было ясно, что программа является анонимной транзакцией. Введение этой директивы — единственное изменение в коде PL/SQL для поддержки анонимных транзакций. Все остальное — команды `COMMIT`

и ROLLBACK, а также команды DML — остается прежним. Автономная транзакция просто изменяет их область действия и видимость при выполнении в контексте автономной транзакции, и вы должны включить команду COMMIT или ROLLBACK в каждую программу автономной транзакции.

Правила и ограничения на использование автономных транзакций

Включить в программный код директиву AUTONOMOUS_TRANSACTION несложно, однако при ее использовании действует целый ряд правил и ограничений.

- Если автономная транзакция пытается обратиться к ресурсу, заблокированному главной транзакцией (приостановленной до завершения автономной программы), происходит взаимная блокировка (deadlock). Простой пример: программа обновляет таблицу, а затем, не зафиксировав изменения, вызывает автономную процедуру, которая должна внести изменения в ту же таблицу:

```
/* Файл в Сети: autondlock.sql */
PROCEDURE update_salary (dept_in IN NUMBER)
IS
    PRAGMA AUTONOMOUS_TRANSACTION;

    CURSOR myemps IS
        SELECT empno FROM emp
        WHERE deptno = dept_in
        FOR UPDATE NOWAIT;
BEGIN
    FOR rec IN myemps
    LOOP
        UPDATE emp SET sal = sal * 2
        WHERE empno = rec.empno;
    END LOOP;
    COMMIT;
END;
BEGIN
    UPDATE emp SET sal = sal * 2;
    update_salary (10);
END;
```

Результат не радует:

```
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

Одна директива не может определить как автономные все подпрограммы пакета или все методы объектного типа. Эту директиву нужно явно включать в каждую программу. Соответственно, на основании только спецификации пакета невозможно определить, какие из программ работают в режиме автономной транзакции.

- Чтобы выход из программы с автономной транзакцией, выполнившей как минимум одну команду INSERT, UPDATE, MERGE или DELETE, произошел без ошибок, необходимо выполнить явную фиксацию или откат транзакции. Если в программе содержится незавершенная транзакция, то исполняющее ядро инициирует исключение и производит откат незафиксированной транзакции.
- Команды COMMIT и ROLLBACK завершают активную автономную транзакцию, но не программу, в которой она содержится. Поэтому блок с автономной транзакцией может содержать несколько команд COMMIT и ROLLBACK.
- При выполнении отката до точки сохранения эта точка должна быть определена в текущей транзакции. В частности, нельзя выполнить откат из автономной транзакции до точки сохранения, определенной в главной транзакции.

- Параметр `TRANSACTIONS` в инициализационном файле Oracle определяет допустимое количество выполняемых одновременно автономных транзакций для одного сеанса. Если в приложении используется множество программ, выполняемых в режиме автономной транзакции, этот предел может быть превышен, что приведет к выдаче исключения. В этом случае значение параметра `TRANSACTIONS` следует увеличить. По умолчанию оно равно 75.

Область видимости транзакций

После выполнения в автономной транзакции команд `COMMIT` или `ROLLBACK` изменения по умолчанию немедленно становятся видимыми в главной транзакции. А если вы предпочитаете скрыть их от последней? Да, изменения должны быть сохранены или отменены, но информация об этом не должна становиться доступной в главной транзакции. Для этой цели используется следующая модификация команды `SET TRANSACTION`:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

По умолчанию для транзакции устанавливается уровень изоляции `READ COMMITTED`, при котором зафиксированные изменения тут же становятся видимыми главной транзакции. Как обычно, команда `SET TRANSACTION` должна выполняться до начала транзакции (то есть перед первой командой SQL). Кроме того, данная установка влияет на весь сеанс, а не только на текущую программу. Сценарий `autonserial.sql` на сайте книги демонстрирует уровень изоляции `SERIALIZABLE` в действии.

В каких случаях следует применять автономные транзакции

Прежде всего следует понять общее правило: программный модуль определяется как автономный, если выполняемые в нем изменения должны быть изолированы от контекста транзакции вызывающего кода. Приведем несколько типичных случаев применения автономных транзакций.

- **Механизм протоколирования.** Требуется записать информацию об ошибке в таблицу базы данных, а также выполнить откат внутренней транзакции, приведшей к ошибке. При этом вы не хотите, чтобы из таблицы-журнала были удалены и другие записи об ошибках. Сделайте внутреннюю транзакцию автономной! Вероятно, это самая распространенная причина для использования автономных транзакций в коде PL/SQL.
- **Фиксация и откат в триггерах базы данных.** Определив триггер как автономную транзакцию, все выполненные им изменения можно фиксировать или отменять, и это никак не отразится на транзакции, приведшей к срабатыванию триггера. Для чего это может быть нужно? Например, в триггере базы данных можно выполнить действие, которое не зависит от статуса транзакции, вызвавшей срабатывание триггера. Предположим, вы хотите отслеживать все действия с таблицей независимо от того, было это действие завершено или нет. А может, вы хотите отслеживать действия, завершившиеся неудачей. Примеры использования этого приема представлены в сценариях `autontrigger*.sql` на сайте книги.
- **Многократно используемые компоненты приложения.** Для программ этого типа возможность определять автономные транзакции жизненно необходима. В современных системах, особенно в распределенном и многоуровневом мире Интернета, необходимы независимые программные единицы, выполняемые без каких-либо побочных эффектов для среды вызова. Автономные транзакции играют важную роль в этой области.

- **Предотвращение каскадных триггерных ошибок.** Такие ошибки возникают в ситуации, когда триггер уровня строки таблицы пытается читать или записывать данные в таблицу, из которой он сработал. Но если преобразовать триггер в автономную транзакцию, включив директиву `PRAGMA AUTONOMOUS_TRANSACTION` и закрепив изменения в теле триггера, код последнего сможет запрашивать содержимое таблицы, но при этом будет видеть только уже закрепленные изменения. Иначе говоря, в таблице не будут видны изменения, внесение которых привело к срабатыванию триггера. Кроме того, код триггера не сможет изменять содержимое таблицы.
- **Вызов пользовательских функций в коде SQL, изменяющем таблицы.** Oracle позволяет вызывать в командах SQL пользовательские функции — при условии, что функция не обновляет базу данных (и с некоторыми дополнительными условиями). Но если функция будет определена как автономная транзакция, вы можете выполнять в ней операции вставки, обновления и удаления в ходе запроса. Данная возможность продемонстрирована в сценарии `trcfunc.sql` на сайте книги.
- **Счетчик попыток.** Допустим, вы хотите предоставить пользователю N попыток обращения к ресурсу, причем количество попыток должно сохраняться между подключениями к базе данных. Для этого необходима команда `COMMIT`, независимая от транзакции. Примеры представлены в файлах `retry.pkg` и `retry.tst` на сайте книги.

Создание механизма автономного протоколирования

Во многих приложениях возникает необходимость в ведении журнала ошибок, происходящих в ходе выполнения транзакций. Самым удобным местом для хранения журнала является таблица базы данных: она всегда доступна и из нее удобно извлекать информацию для анализа средствами SQL.

Но у такого журнала имеется один недостаток: операции записи в журнал становятся частью транзакции. И если будет выполнена команда `ROLLBACK`, из журнала может быть удалена часть записей об ошибках. Конечно, это нежелательно. Можно попытаться с помощью точек сохранения выполнить откат таким образом, чтобы записи журнала оставались нетронутыми, но это крайне неудобно. Автономные транзакции существенно упрощают протоколирование ошибок, делают его более удобным и надежным.

Допустим, таблица-журнал определена следующим образом:

```
/* Файл в Сети: log.pkg */
CREATE TABLE logtab (
    code INTEGER, text VARCHAR2(4000),
    created_on DATE, created_by VARCHAR2(100),
    changed_on DATE, changed_by VARCHAR2(100)
);
```

Таблица будет использоваться для хранения информации об ошибках (функции `SQLCODE` и `SQLERRM`), а также для протоколирования других событий, не связанных с ошибками.

Как пользоваться журналом? Прежде всего рассмотрим неправильное решение:

```
EXCEPTION
    WHEN OTHERS
    THEN
        DECLARE
            v_code PLS_INTEGER := SQLCODE;
            v_msg VARCHAR2(1000) := SQLERRM;
        BEGIN
            INSERT INTO logtab VALUES (
                v_code, v_msg, SYSDATE, USER, SYSDATE, USER);
        END;
    END;
```

Другими словами, никогда не реализуйте механизм протоколирования в разделе исключений и других разделах основного кода приложения. Все операции с таблицей-журналом должны выполняться на отдельном программном уровне (это называется *инкапсуляцией*). Для этого есть несколько важных причин:

- Если структура таблицы однажды изменится, программы, в которых производится протоколирование ошибок, не пострадают.
- С журналом удобнее работать, а операции имеют более общий характер.
- Подпрограмма может быть реализована в виде автономной транзакции.

Наш очень простой пакет протоколирования состоит всего из двух процедур:

```
PACKAGE log
IS
    PROCEDURE putline (code_in IN INTEGER, text_in IN VARCHAR2);
    PROCEDURE saveline (code_in IN INTEGER, text_in IN VARCHAR2);
END;
```

Чем `putline` отличается от `saveline`? Процедура `log.saveline` является автономной транзакцией, тогда как `log.putline` просто вставляет строку в таблицу. Вот тело пакета:

```
/* Файл в Сети: log.pkg */
PACKAGE BODY log
IS
    PROCEDURE putline (
        code_in IN INTEGER, text_in IN VARCHAR2)
    IS
    BEGIN
        INSERT INTO logtab
            VALUES (
                code_in,
                text_in,
                SYSDATE,
                USER,
                SYSDATE,
                USER
            );
    END;

    PROCEDURE saveline (
        code_in IN INTEGER, text_in IN VARCHAR2)
    IS
    BEGIN
        PRAGMA AUTONOMOUS_TRANSACTION;
        putline (code_in, text_in);
        COMMIT;
    EXCEPTION WHEN OTHERS THEN ROLLBACK;
    END;
END;
```

Несколько замечаний по поводу реализации, которые помогут вам лучше понять происходящее:

- Процедура `putline` выполняет обыкновенную вставку. Вероятно, вы захотите добавить в нее раздел обработки исключений, если пакет `log` будет взят за основу подсистемы протоколирования ошибок в реальном приложении.
- Процедура `saveline` вызывает процедуру `putline`, но делает это в контексте автономной транзакции.

При наличии описываемого пакета обработка ошибок значительно упрощается:

```
EXCEPTION
  WHEN OTHERS
  THEN
    log.saveline (SQLCODE, SQLERRM);
END;
```

Только и всего. Разработчикам не нужно задумываться о структуре таблицы журнала; более того, они могут вообще не знать, что запись производится в таблицу базы данных. А поскольку транзакция автономна, то что бы ни происходило в приложении — журнал останется в целости и сохранности.

15

Выборка данных

Одной из важнейших характеристик PL/SQL является тесная интеграция с базой данных Oracle в отношении как изменения данных в таблицах, так и выборки данных из таблиц. В этой главе рассматриваются элементы PL/SQL, связанные с выборкой информации из базы данных и ее обработкой в программах PL/SQL.

При выполнении команды SQL из PL/SQL PCYБД Oracle назначает ей приватную рабочую область, а некоторые данные записывает в системную глобальную область (SGA, System Global Area). В приватной рабочей области содержится информация о команде SQL и набор данных, возвращаемых или обрабатываемых этой командой. PL/SQL предоставляет программистам несколько механизмов доступа к этой рабочей области и содержащейся в ней информации; все они так или иначе связаны с определением курсоров и выполнением операций с ними.

- **Неявные курсоры.** Команда `SELECT...INTO` считывает одну строку данных и присваивает ее в качестве значения локальной переменной программы. Это простейший (и зачастую наиболее эффективный) способ доступа к данным, но он часто ведет к написанию сходных и даже одинаковых SQL-команд `SELECT` во многих местах программы.
- **Явные курсоры.** Запрос можно явно объявить как курсор в разделе объявлений локального блока или пакета. После этого такой курсор можно будет открывать и выбирать из него данные в одной или нескольких программах, причем возможности управления явным курсором шире, чем у неявного.
- **Курсорные переменные.** Курсорные переменные (в объявлении которых задается тип `REF CURSOR`) позволяют передавать из программы в программу *указатель* на результирующий набор строк запроса. Любая программа, для которой доступна такая переменная, может открыть курсор, извлечь из него необходимые данные и закрыть его.
- **Курсорные выражения.** Ключевое слово `CURSOR` превращает команду `SELECT` в набор `REF CURSOR`, который может использоваться совместно с табличными функциями для повышения производительности приложения.
- **Динамические SQL-запросы.** Oracle позволяет динамически конструировать и выполнять запросы с использованием либо встроенного динамического SQL (NDS — см. главу 16), либо программ пакета `DMBS_SQL`. Этот встроенный пакет описывается в документации Oracle, а также в книге *Oracle Built-in Packages* (O'Reilly).

В этой главе рассказано о неявных и явных курсорах, курсорных переменных и выражениях.

Основные принципы работы с курсорами

Курсор проще всего представить себе как указатель на таблицу в базе данных. Например, следующее объявление связывает всю таблицу `employee` с курсором `employee_cur`:

```
CURSOR employee_cur IS SELECT * FROM employee;
```

Объявленный курсор можно открыть:

```
OPEN employee_cur;
```

Далее из него можно выбирать строки:

```
FETCH employee_cur INTO employee_rec;
```

Завершив работу с курсором, его следует закрыть:

```
CLOSE employee_cur;
```

В этом случае каждая выбранная из курсора запись представляет строку таблицы `employee`. Однако с курсором можно связать любую допустимую команду `SELECT`. В следующем примере в объявлении курсора объединяются три таблицы:

```
DECLARE
  CURSOR joke_feedback_cur
  IS
    SELECT J.name, R.laugh_volume, C.name
      FROM joke J, response R, comedian C
     WHERE J.joke_id = R.joke_id
          AND R.joker_id = C.joker_id;
BEGIN
  ...
END;
```

В данном случае курсор действует не как указатель на конкретную таблицу базы данных — он указывает на виртуальную таблицу или неявное представление, определяемое командой `SELECT`. (Такая таблица называется виртуальной, потому что команда `SELECT` генерирует данные с табличной структурой, но эта таблица существует только временно, пока программа работает с возвращенными командой данными.) Если тройное объединение возвращает таблицу из 20 строк и 3 столбцов, то курсор действует как указатель на эти 20 строк.

Терминология

В PL/SQL имеется множество возможностей выполнения команд SQL, и все они реализованы в программах как курсоры того или иного типа. Прежде чем приступить к их освоению, необходимо познакомиться с методами выборки данных и используемой при этом терминологией.

- **Статический SQL.** Команда SQL называется *статической*, если она полностью определяется во время компиляции программы.
- **Динамический SQL.** Команда SQL называется *динамической*, если она строится и выполняется на стадии выполнения программы, так что в программном коде нет ее фиксированного объявления. Для динамического выполнения команд SQL могут использоваться программы встроенного пакета `DBMS_SQL` (имеющегося во всех версиях Oracle) или встроенный динамический SQL.
- **Результирующий набор строк.** Набор строк с результирующими данными, удовлетворяющими критериям, определяемым командой SQL. Результирующий набор кэшируется в системной глобальной области с целью ускорения чтения и модификации его данных.
- **Неявный курсор.** При каждом выполнении команды DML (`INSERT`, `UPDATE`, `MERGE` или `DELETE`) или команды `SELECT INTO`, возвращающей строку из базы данных прямо

в структуру данных программы, PL/SQL создает неявный курсор. Курсор этого типа называется *неявным*, поскольку Oracle автоматически выполняет многие связанные с ним операции, такие как открытие, выборка данных и даже закрытие.

- **Явный курсор.** Команда `SELECT`, явно определенная в программе как курсор. Все операции с явным курсором (открытие, выборка данных, закрытие и т. д.) в программе должны выполняться явно. Как правило, явные курсоры используются для выборки из базы данных набора строк с использованием статического SQL.
- **Курсорная переменная.** Объявленная программистом переменная, указывающая на объект курсора в базе данных. Ее значение (то есть указатель на курсор или результирующий набор строк) во время выполнения программы может меняться, как у всех остальных переменных. В разные моменты времени курсорная переменная может указывать на разные объекты курсора. Курсорную переменную можно передать в качестве параметра процедуре или функции. Такие переменные очень полезны для передачи результирующих наборов из программ PL/SQL в другие среды (например, Java или Visual Basic).
- **Атрибут курсора.** Атрибут курсора имеет форму `%имя_атрибута` и добавляется к имени курсора или курсорной переменной. Это что-то вроде внутренней переменной Oracle, возвращающей информацию о состоянии курсора — например о том, открыт ли курсор, или сколько строк из курсора вернул запрос. У явных и неявных курсоров и в динамическом SQL в атрибутах курсоров существуют некоторые различия, которые рассматриваются в этой главе.
- **SELECT FOR UPDATE.** Разновидность обычной команды `SELECT`, устанавливающая блокировку на каждую возвращаемую запросом строку данных. Пользоваться ею следует только в тех случаях, когда нужно «зарезервировать» запрошенные данные, чтобы никто другой не мог изменить их, пока с ними работаете вы.
- **Пакетная обработка.** В Oracle8i и выше PL/SQL поддерживает запросы с секцией `BULK COLLECT`, позволяющей за один раз выбрать из базы данных более одной строки.

Типичные операции с запросами и курсорами

Независимо от типа курсора процесс выполнения команд SQL всегда состоит из одних и тех же действий. В одних случаях PL/SQL производит их автоматически, а в других, как, например, при использовании явного курсора, они явно организуются программистом.

- **Разбор.** Первым шагом при обработке команды SQL должен быть ее *разбор* (синтаксический анализ), то есть проверка ее корректности и формирование плана выполнения (с применением оптимизации по синтаксису или по стоимости в зависимости от того, какое значение параметра `OPTIMIZER_MODE` задал администратор базы данных).
- **Привязка.** *Привязкой* называется установление соответствия между значениями программы и параметрами команды SQL. Для статического SQL привязка производится ядром PL/SQL. Привязка параметров в динамическом SQL выполняется явно с использованием переменных привязки.
- **Открытие.** При открытии курсора определяется результирующий набор строк команд SQL, для чего используются переменные привязки. Указатель активной или текущей строки указывает на первую строку результирующего набора. Иногда явное открытие курсора не требуется; ядро PL/SQL выполняет эту операцию автоматически (так происходит в случае применения неявных курсоров и встроенного динамического SQL).
- **Выполнение.** На этой стадии команда выполняется ядром SQL.

- **Выборка.** Выборка очередной строки из результирующего набора строк курсора осуществляется командой `FETCH`. После каждой выборки PL/SQL перемещает указатель на одну строку вперед. Работая с явными курсорами, помните, что и после завершения перебора всех строк можно снова и снова выполнять команду `FETCH`, но PL/SQL ничего не будет делать (и не станет инициировать исключение) — для выявления этого условия следует использовать атрибуты курсора.
- **Закрытие.** Операция закрывает курсор и освобождает используемую им память. Закрытый курсор уже не содержит результирующий набор строк. Иногда явное закрытие курсора не требуется, последовательность PL/SQL делает это автоматически (для неявных курсоров и встроенного динамического SQL).

На рис. 15.1 показано, как некоторые из этих операций используются для выборки информации из базы данных в программу PL/SQL.

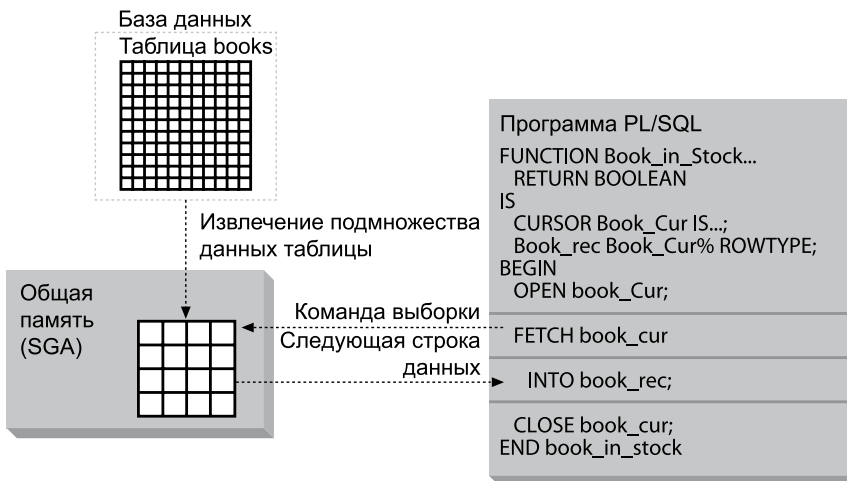


Рис. 15.1. Упрощенная схема выборки данных с использованием курсора

Знакомство с атрибутами курсоров

В этом разделе перечисляются и вкратце описываются атрибуты курсоров. Более подробные описания атрибутов для каждого вида курсоров приводятся в этой главе, а также в главах 14 и 16.

PL/SQL поддерживает шесть атрибутов курсоров, перечисленных в табл. 15.1.

Таблица 15.1. Атрибуты курсоров

| Имя | Что возвращает |
|------------------|---|
| %FOUND | TRUE, если успешно выбрана хотя бы одна строка; в противном случае возвращает FALSE |
| %NOTFOUND | TRUE, если команда не выбрала ни одной строки; в противном случае возвращает FALSE |
| %ROWCOUNT | Количество строк, выбранных из курсора на данный момент времени |
| %ISOPEN | TRUE, если курсор открыт; в противном случае возвращает FALSE |
| %BULK_ROWCOUNT | Количество измененных записей для каждого элемента исходной коллекции, заданной в команде <code>FORALL</code> |
| %BULK_EXCEPTIONS | Информация об исключении для каждого элемента исходной коллекции, заданной в команде <code>FORALL</code> |

Чтобы обратиться к атрибуту курсора, укажите в виде его префикса имя курсора или курсорной переменной и символ %:

имя_курсора%имя_атрибута

В качестве имен неявных курсоров используется префикс `SQL`, например `SQL%NOTFOUND`.

Ниже приведены краткие описания всех атрибутов курсоров.

Атрибут %FOUND

Атрибут `%FOUND` возвращает информацию о состоянии последней операции `FETCH` с курсором. Если последний вызов `FETCH` для курсора вернул строку, то возвращается значение `TRUE`, а если строка не была получена, возвращается `FALSE`.

Если курсор еще не был открыт, база данных инициирует исключение `INVALID_CURSOR`.

В следующем примере я перебираю все строки курсора `caller_cur`, присваиваю все звонки, введенные до сегодняшнего дня для конкретного позвонившего, после чего перехожу к следующей записи. При достижении последней записи атрибут `%FOUND` явного курсора возвращает `FALSE`, и выполнение простого цикла прерывается. Атрибут `%FOUND` неявного курсора также проверяется после команды `UPDATE`:

```
FOR caller_rec IN caller_cur
LOOP
    UPDATE call
    SET caller_id = caller_rec.caller_id
    WHERE call_timestamp < SYSDATE;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('Calls updated for ' || caller_rec.caller_id);
    END IF;
END LOOP;
```

Атрибут %NOTFOUND

Атрибут `%NOTFOUND` по смыслу противоположен `%FOUND`: он возвращает `TRUE`, если последняя операция `FETCH` с курсором не вернула строки (как правило, из-за того, что последняя строка уже была прочитана). Если курсор не может вернуть строку из-за ошибки, инициируется соответствующее исключение.

Если курсор еще не был открыт, база данных инициирует исключение `INVALID_CURSOR`.

В каких случаях следует использовать `%FOUND`, а когда предпочтение отдается `%NOTFOUND`?

Используйте ту формулировку, которая более естественно подходит для вашего кода.

В предыдущем примере для выхода из цикла использовалась следующая команда:

```
EXIT WHEN NOT caller_cur%FOUND;
```

Альтернативная — и возможно, более понятная — формулировка могла бы использовать `%NOTFOUND`:

```
EXIT WHEN caller_cur%NOTFOUND;
```

Атрибут %ROWCOUNT

Атрибут `%ROWCOUNT` возвращает количество записей, прочитанных из курсора к моменту запроса атрибута. При исходном открытии курсора атрибут `%ROWCOUNT` равен 0. При обращении к атрибуту `%ROWCOUNT` курсора, который еще не был открыт, инициируется исключение `INVALID_CURSOR`. После выборки каждой записи `%ROWCOUNT` увеличивается на единицу.

Используйте атрибут `%ROWCOUNT` для проверки того, что программа прочитала (или обновила в случае `DML`) нужное количество строк, или для прерывания выполнения после заданного числа итераций. Пример:

```
BEGIN
  UPDATE employees SET last_name = 'FEUERSTEIN';
  DBMS_OUTPUT.PUT_LINE (SQL%ROWCOUNT);
END;
```

Атрибут %ISOPEN

Атрибут %ISOPEN возвращает TRUE, если курсор открыт; в противном случае возвращается FALSE.

Приведем типичный пример использования этого атрибута, который гарантирует, что курсор не остается открытым, если в программе произойдет что-то непредвиденное:

```
DECLARE
  CURSOR happiness_cur IS SELECT simple_delights FROM ...;
BEGIN
  OPEN happiness_cur;
  ...
  IF happiness_cur%ISOPEN THEN ...
EXCEPTION
  WHEN OTHERS THEN
    IF happiness_cur%ISOPEN THEN
      close happiness_cur;
    END IF;
END;
```

Атрибут %BULK_ROWCOUNT

Атрибут %BULK_ROWCOUNT, предназначенный для использования с командой FORALL, возвращает количество строк, обработанных при каждом выполнении DML. Этот атрибут обладает семантикой ассоциативного массива (см. главу 21).

Атрибут %BULK_EXCEPTIONS

Атрибут %BULK_EXCEPTIONS, также предназначенный для использования с командой FORALL, возвращает информацию об исключении, которое может быть инициировано при каждом выполнении DML. Этот атрибут (см. главу 21) обладает семантикой ассоциативного массива записей.



На атрибуты курсоров можно ссылаться в коде PL/SQL, как показано в приведенных примерах, но вы не сможете напрямую обращаться к ним в команде SQL. Например, при попытке использовать атрибут %ROWCOUNT в секции WHERE команды SELECT:

```
SELECT caller_id, company_id FROM caller
WHERE company_id = company_cur%ROWCOUNT;
```

компилятор выдает ошибку PLS-00229: Attribute expression within SQL expression.

Ссылки на переменные PL/SQL в курсорах

Поскольку курсор должен быть связан с командой SELECT, в нем всегда присутствует хотя бы одна ссылка на таблицу базы данных; по ней (и по содержимому условия WHERE) Oracle определяет, какие строки будут возвращены в составе активного набора. Однако это не означает, что команда SELECT может возвращать информацию только из базы данных.

Список выражений, задаваемых между ключевыми словами SELECT и FROM, называется *списком выборки*. Во встроенном SQL список выборки может содержать столбцы и выражения (вызовы SQL-функций для этих столбцов, константы и т. д.). В PL/SQL список выборки обычно содержит переменные PL/SQL и сложные выражения.

На локальные данные программы PL/SQL (переменные и константы), а также на переменные привязки хост-среды можно ссылаться в предложениях `WHERE`, `GROUP BY` и `HAVING` команды `SELECT` курсора. Ссылки на переменные PL/SQL можно (и должно) *уточнять* именем ее области видимости (именем процедуры, именем пакета и т. д.), особенно в командах SQL.

Выбор между явным и неявным курсорами

Все последние годы знатоки Oracle (включая и авторов данной книги) убежденно доказывали, что для однострочной выборки данных никогда не следует использовать неявные курсоры. Это мотивировалось тем, что неявные курсоры, соответствуя стандарту ISO, всегда выполняют две выборки, из-за чего они уступают по эффективности явным курсорам.

Так утверждалось и в первых двух изданиях этой книги, но пришло время нарушить эту традицию (вместе со многими другими). Начиная с Oracle8, в результате целенаправленных оптимизаций неявные курсоры выполняются даже *эффективнее* эквивалентных явных курсоров.

Означает ли это, что теперь всегда лучше пользоваться неявными курсорами? Вовсе нет. В пользу применения явных курсоров существуют убедительные доводы.

- В некоторых случаях явные курсоры эффективнее неявных. Часто выполняемые критические запросы лучше протестировать в обеих формах, чтобы точно выяснить, как лучше выполнять каждый из них в каждом конкретном случае.
- Явными курсорами проще управлять из программы. Например, если строка не найдена, Oracle не иницирует исключение, а просто принудительно завершает выполняемый блок.

Поэтому вместо формулировки «явный или неявный?» лучше спросить: «инкапсулированный или открытый?» И ответ будет таким: всегда инкапсулируйте однострочные запросы, скрывая их за интерфейсом функции (желательно пакетной) и возвращая данные через `RETURN`.

Не жалейте времени на инкапсуляцию запросов в функциях, желательно пакетных. Это позволит вам и всем остальным разработчикам вашей группы просто вызвать функцию, когда появится необходимость в данных. Если Oracle изменит правила обработки запросов, а ваши предыдущие наработки станут бесполезными, достаточно будет изменить реализацию всего одной функции.

Работа с неявными курсорами

При каждом выполнении команды DML (`INSERT`, `UPDATE`, `MERGE` или `DELETE`) или команды `SELECT INTO`, возвращающей строку из базы данных в структуру данных программы, PL/SQL автоматически создает для нее курсор. Курсор этого типа называется неявным, поскольку Oracle автоматически выполняет многие связанные с ним операции, такие как выделение курсора, его открытие, выборку строк и т. д.



Выполнение команд DML с помощью неявных курсоров рассматривается в главе 14. В этой главе рассматриваются только неявные запросы SQL.

Неявный курсор — это команда `SELECT`, обладающая следующими характеристиками:

- Команда `SELECT` определяется в исполняемом разделе блока, а не в разделе объявлений, как явные курсоры.

- В команде содержится предложение INTO (или BULK COLLECT INTO), которое относится к языку PL/SQL (а не SQL) и представляет собой механизм передачи данных из базы в локальные структуры данных PL/SQL.
- Команду SELECT не нужно открывать, выбирать из нее данные и закрывать; все эти операции осуществляются автоматически.

Общая структура неявного запроса выглядит так:

```
SELECT список_столбцов  
[BULK COLLECT] INTO PL/SQL список_переменных  
...продолжение команды SELECT...
```

Если в программе используется неявный курсор, Oracle автоматически открывает его, выбирает строки и закрывает; над этими операциями программист не властен. Однако он может получить информацию о последней выполненной команде SQL, анализируя значения атрибутов неявного курсора SQL, как рассказывается далее в этой главе.



В следующих разделах, говоря о неявных курсорах, мы будем иметь в виду команду SELECT INTO, которая извлекает (или пытается извлечь) одну строку данных. В главе 21 обсуждается ее разновидность SELECT BULK COLLECT INTO, которая позволяет с помощью одного неявного запроса получить несколько строк данных.

Примеры неявных курсоров

Неявные курсоры часто используются для поиска данных на основе значений первичного ключа. В следующем примере выполняется поиск названия книги по ее коду ISBN:

```
DECLARE  
  l_title books.title%TYPE;  
BEGIN  
  SELECT title  
    INTO l_title  
    FROM books  
   WHERE isbn = '0-596-00121-5';
```

После того как название книги будет выбрано и присвоено локальной переменной l_title, с последней можно работать как с любой другой переменной: изменять ее значение, выводить на экран или передавать для обработки другой программе PL/SQL.

Пример неявного курсора, извлекающего из таблицы строку данных и помещающего ее в запись программы:

```
DECLARE  
  l_book books%ROWTYPE;  
BEGIN  
  SELECT *  
    INTO l_book  
    FROM books  
   WHERE isbn = '0-596-00121-5';
```

Из запроса также можно получить информацию уровня групп. Например, следующий запрос вычисляет и возвращает сумму окладов по отделу. И снова PL/SQL создает для него неявный курсор:

```
SELECT SUM (salary)  
  INTO department_total  
  FROM employees  
 WHERE department_id = 10;
```

Благодаря тесной интеграции PL/SQL с базой данных Oracle с помощью запросов из базы данных можно извлекать и сложные типы данных, такие как объекты и коллекции. Все эти примеры демонстрируют применение неявных запросов для выборки данных

одной строки. Если вы хотите получить более одной строки, используйте либо явный курсор, либо конструкцию **BULK COLLECT INTO** (см. главу 21).



Как упоминалось ранее, я рекомендую всегда «скрывать» однострочные запросы за функциональным интерфейсом. Эта концепция подробно рассматривается в разделе «Выбор между явным и неявным курсорами» этой главы.

Обработка ошибок при использовании неявных курсоров

Команда **SELECT**, выполняемая как неявный курсор, представляет собой «черный ящик». Она передается в базу данных и возвращает одну строку информации. Что происходит с курсором, как он открывается, получает данные и закрывается, вы не знаете. Также приходится смириться с тем, что в двух стандартных ситуациях при выполнении команды **SELECT** автоматически инициируются исключения:

- По запросу не найдено ни одной строки. В этом случае Oracle инициирует исключение **NO_DATA_FOUND**.
- Команда **SELECT** вернула несколько строк. В этом случае Oracle инициирует исключение **TOO_MANY_ROWS**.

В каждом из этих случаев (как и при возникновении любого другого исключения, инициированного при выполнении команды **SQL**) выполнение текущего блока прерывается, и управление передается в раздел исключений. Этим процессом вы не управляете; у вас даже нет возможности сообщить Oracle, что данный запрос может не вернуть ни одной строки, и это не является ошибкой. Таким образом, если вы используете неявный курсор, в раздел исключений обязательно нужно включить код перехвата и обработки этих двух исключений (а возможно, и других исключений, как того требует логика программы).

В следующем блоке кода производится поиск названия книги по ее коду ISBN с обработкой возможных исключений:

```
DECLARE
    l_isbn books.isbn%TYPE := '0-596-00121-5';
    l_title books.title%TYPE;
BEGIN
    SELECT title
    INTO l_title
    FROM books
    WHERE isbn = l_isbn;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('Неизвестная книга: ' || l_isbn);
    WHEN TOO_MANY_ROWS
    THEN
        /* Пакет определяется в errpkg.pkg */
        errpkg.record_and_stop ('Нарушение целостности данных для: ' || l_isbn);
        RAISE;
END;
```

При работе с неявным курсором программист может сделать некоторые рискованные предположения об извлекаемых данных. Приведем пару примеров:

- В таблице не может быть более одной книги с указанным кодом ISBN — ведь это гарантируется заданным для таблицы ограничением.
- В таблице обязательно имеется строка с данными о содержащейся в ней книге, поэтому не стоит беспокоиться об исключении **NO_DATA_FOUND**.

Последствия таких предположений часто оказываются плачевными, поскольку программисты не включают в программы соответствующие обработчики исключений для неявных запросов.

Конечно, в настоящий момент и при текущем состоянии данных запрос может вернуть ровно одну строку. Но если данные изменятся, запрос, к примеру, команда `SELECT` может вернуть две строки вместо одной, программа выдаст исключение, которое не будет обработано — и это может создать проблемы в коде.

Как правило, при использовании неявных курсоров желательно всегда включать в программу обработчики исключений `NO_DATA_FOUND` и `TOO_MANY_ROWS`. В обобщенной формулировке можно сказать, что в программе должны присутствовать обработчики всех исключений, которые в ней могут произойти. А вот действия, выполняемые этими обработчиками, могут быть самыми разными. Возьмем все тот же код, извлекающий название книги по заданному коду ISBN. Ниже он реализован в виде функции с двумя обработчиками ошибок. Обработчик `NO_DATA_FOUND` возвращает значение, а обработчик `TOO_MANY_ROWS` записывает ошибку в журнал и повторно иницирует исключение, прерывая работу функции. (О пакете `errpkg.pkg` более подробно рассказано в главе 6.)

```
FUNCTION book_title (isbn_in  IN    books.isbn%TYPE
)
RETURN books.title%TYPE
IS
    return_value  book.title%TYPE;
BEGIN
    SELECT title
    INTO return_value
    FROM books
    WHERE isbn = isbn_in;

    RETURN return_value;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        RETURN NULL;
    WHEN TOO_MANY_ROWS
    THEN
        errpkg.record_and_stop ('Нарушение целостности данных для: '
                                || isbn_in);
        RAISE;
END;
```

Почему эти два обработчика ведут себя по-разному? Дело в том, что функция должна вернуть название книги, которое никогда не может быть представлено значением `NULL`. Для проверки используется код ISBN («существует ли книга с данным кодом ISBN?»), поэтому если книга по ISBN не найдена, функция не должна инициировать исключение. В этом нет ничего плохого. Например, логика программы может быть такой: «если книги с заданным ISBN не существует, используем его для новой книги», а возможная реализация может выглядеть так:

```
IF book_title ('0-596-00121-7') IS NULL
THEN ...
```

Иначе говоря, то, что запрос не вернул ни одной строки, не всегда свидетельствует об ошибке.

С другой стороны, если запрос сгенерировал исключение `TOO_MANY_ROWS`, мы сталкиваемся с настоящей проблемой: в базе не может быть двух книг с одинаковыми кодами ISBN. Поэтому в данном случае информация об ошибке записывается в журнал, а программа прекращает свою работу.

Атрибуты неявных курсоров

Для получения информации о последнем запросе, выполненном с помощью неявного курсора, Oracle предоставляет программисту четыре атрибута, перечисленных в табл. 15.2. Поскольку неявный курсор не имеет имени, оно заменяется ключевым словом SQL.

Таблица 15.2. Атрибуты неявных курсоров

| Имя | Что возвращает |
|--------------|--|
| SQL%FOUND | TRUE, если команда успешно выбрала одну строку (или несколько строк для запросов BULK COLLECT INTO). В противном случае возвращается FALSE (в этом случае также генерируется исключение NO_DATA_FOUND) |
| SQL%NOTFOUND | TRUE, если команда не выбрала ни одной строки (в этом случае также генерируется исключение NO_DATA_FOUND). В противном случае возвращается FALSE |
| SQL%ROWCOUNT | Количество строк, выбранных из заданного курсора до настоящего момента. Для SELECT INTO возвращается 1, если строка найдена, или 0, если база данных генерирует исключение NO_DATA_FOUND |
| SQL%ISOPEN | FALSE для неявных курсоров, поскольку Oracle закрывает и открывает неявные курсоры автоматически |

Все атрибуты неявных курсоров возвращают NULL, если в сеансе еще не выполнялся ни один неявный курсор. В противном случае значения атрибутов всегда относятся к последней выполненной команде SQL независимо от того, в каком блоке или программе она выполнялась. За дополнительной информацией об этом поведении обращайтесь к разделу «Атрибуты курсора для операций DML» главы 14. Для тестирования этих значений также можно воспользоваться сценарием `query_implicit_attributes.sql` на сайте книги.

Чтобы этот момент стал более понятным, рассмотрим следующие две программы:

```

PROCEDURE remove_from_circulation
    (isbn_in in books.isbn%TYPE)
IS
BEGIN
    DELETE FROM book WHERE isbn = isbn_in;
END;
PROCEDURE show_book_count
IS
    l_count    INTEGER;
BEGIN
    SELECT COUNT (*)
        INTO l_count
        FROM books;

    -- Такой книги нет!
    remove_from_circulation ('0-000-00000-0');
```

Независимо от количества строк данных в таблице `books` результат всегда будет равен 0. Поскольку после команды `SELECT INTO` вызывается процедура `remove_from_circulation`, атрибут `SQL%ROWCOUNT` возвращает информацию о результате немыслимой команды `DELETE`, а вовсе не количество строк, возвращаемых запросом.

Если вам нужна гарантия того, что значения атрибутов относятся к заданной команде SQL, сохраняйте эти атрибуты в локальных переменных сразу же после ее выполнения:

```

PROCEDURE show_book_count
IS
    l_count    INTEGER;
    l_numfound PLS_INTEGER;
BEGIN
    SELECT COUNT (*)
```

```

        INTO l_count
        FROM books;

-- Копирование значения атрибута
l_numfound := SQL%ROWCOUNT;

-- Такой книги нет!
remove_from_circulation ('0-000-00000-0');

-- Теперь можно вернуться к предыдущему значению атрибута.
DBMS_OUTPUT.put_line (l_numfound);
END;
```

Работа с явными курсорами

Явный курсор представляет собой команду `SELECT`, явно определенную в разделе объявлений программы. При объявлении явного курсора ему присваивается имя. Для команд `INSERT`, `UPDATE`, `MERGE` и `DELETE` явные курсоры определяться не могут.

Определив команду `SELECT` как явный курсор, программист получает контроль над основными стадиями выборки информации из базы данных. Он определяет, когда открыть курсор (`OPEN`), когда выбрать из него строки (`FETCH`), сколько выбрать строк и когда закрыть курсор с помощью команды `CLOSE`. Информация о текущем состоянии курсора доступна через его атрибуты. Именно высокая детализация контроля делает явные курсоры бесценным инструментом для программиста.

Рассмотрим пример:

```

1  FUNCTION jealousy_level (
2      NAME_IN  IN  friends.NAME%TYPE) RETURN NUMBER
3  AS
4      CURSOR jealousy_cur
5      IS
6          SELECT location FROM friends
7              WHERE NAME = UPPER (NAME_IN);
8
9      jealousy_rec  jealousy_cur%ROWTYPE;
10     retval        NUMBER;
11 BEGIN
12     OPEN jealousy_cur;
13
14     FETCH jealousy_cur INTO jealousy_rec;
15
16     IF jealousy_cur%FOUND
17     THEN
18         IF jealousy_rec.location = 'PUERTO RICO'
19         THEN retval := 10;
20         ELSIF jealousy_rec.location = 'CHICAGO'
21         THEN retval := 1;
22         END IF;
23     END IF;
24
25     CLOSE jealousy_cur;
26
27     RETURN retval;
28 EXCEPTION
29     WHEN OTHERS THEN
30         IF jealousy_cur%ISOPEN THEN
31             CLOSE jealousy_cur;
32         END IF;
33 END;
```

В этом блоке PL/SQL выполняются следующие операции с курсором.

| Строки | Действие |
|--------|---|
| 4–7 | Объявление курсора |
| 9 | Объявление записи на основе курсора |
| 12 | Открытие курсора |
| 14 | Выборка одной строки из курсора |
| 16 | Проверка атрибута курсора с целью узнать, найдена ли строка |
| 18–22 | Анализ содержимого выбранной строки |
| 25 | Закрытие курсора |
| 28–32 | Страховочный код на случай непредвиденных обстоятельств |

В нескольких ближайших разделах подробно рассматривается каждая из перечисленных операций. Термин «курсор» в них относится к явным курсорам, если только в тексте явно не указано обратное.

Объявление явного курсора

Чтобы получить возможность использовать явный курсор, его необходимо объявить в разделе объявлений блока PL/SQL или пакета:

```
CURSOR имя_курсора [ ( [ параметр [, параметр ...] ] ) ]
  [ RETURN спецификация_return ]
  IS команда_SELECT
    [FOR UPDATE [OF [список_столбцов]]];
```

Здесь *имя_курсора* — имя объявляемого курсора; *спецификация_return* — необязательная секция RETURN; *команда_SELECT* — любая допустимая SQL-команда SELECT. Курсору также могут передаваться *параметры* (см. далее раздел «Параметры курсора»). Наконец, после команды SELECT...FOR UPDATE можно задать *список_столбцов* для обновления (также см. далее). После объявления курсор открывается командой OPEN, а выборка строк из него осуществляется командой FETCH.

Несколько примеров объявлений явных курсоров.

- **Курсор без параметров.** Результирующим набором строк этого курсора является набор идентификаторов компаний, выбранных из всех строк таблицы:

```
CURSOR company_cur IS
  SELECT company_id FROM company;
```

- **Курсор с параметрами.** Результирующий набор строк этого курсора содержит единственную строку с именем компании, соответствующим значению переданного параметра:

```
CURSOR name_cur (company_id_in IN NUMBER)
  IS
  SELECT name FROM company
  WHERE company_id = company_id_in;
```

- **Курсор с предложением RETURN.** Результирующий набор строк этого курсора содержит все данные таблицы employee для подразделения с идентификатором 10:

```
CURSOR emp_cur RETURN employees%ROWTYPE
  IS
  SELECT * FROM employees
  WHERE department_id = 10;
```

Имя курсора

Имя явного курсора должно иметь длину до 30 символов и соответствовать тем же правилам, что и остальные идентификаторы PL/SQL. Имя курсора не является переменной — это идентификатор указателя на запрос. Имени курсора не присваивается

значение, его нельзя применять в выражениях. Курсор используется только в командах OPEN, CLOSE и FETCH, а также для уточнения атрибута курсора.

Объявление курсора в пакете

Явные курсоры объявляются в разделе объявлений блока PL/SQL. Курсор может объявляться на уровне пакета, но не в конкретной процедуре или функции пакета. О пакетах подробно рассказано в главе 18. Возможно, прежде чем браться за тему объявления курсоров в пакетах, вам стоит просмотреть главу 18, чтобы составить общее представление о пакетах и их структуре. Пример объявления двух курсоров в пакете:

```
PACKAGE book_info
IS
    CURSOR titles_cur
    IS
        SELECT title
        FROM books;

    CURSOR books_cur (title_filter_in IN books.title%TYPE)
    RETURN books%ROWTYPE
    IS
        SELECT *
        FROM books
        WHERE title LIKE title_filter_in;
END;
```

Первый курсор `titles_cur` возвращает только названия книг. Второй, `books_cur`, возвращает все строки таблицы `books`, в которых названия книг соответствуют шаблону, заданному в качестве параметра курсора (например, «Все книги, содержащие строку 'PL/SQL'»). Обратите внимание: во втором курсоре используется секция `RETURN`, которая объявляет структуру данных, возвращаемую командой `FETCH`.

В секции `RETURN` могут быть указаны любые из следующих структур данных:

- Запись, определяемая на основе строки таблицы данных с помощью атрибута `%ROWTYPE`.
- Запись, определяемая на основе другого, ранее объявленного курсора, также с помощью атрибута `%ROWTYPE`.
- Запись, определенная программистом.

Количество выражений в списке выборки курсора должно соответствовать количеству столбцов записи `имя_таблицы%ROWTYPE`, `курсор%ROWTYPE` или `тип_записи`. Типы данных элементов тоже должны быть совместимы. Например, если второй элемент списка выборки имеет тип `NUMBER`, то второй столбец записи в секции `RETURN` не может иметь тип `VARCHAR2` или `BOOLEAN`.

Прежде чем переходить к подробному рассмотрению секции `RETURN` и ее преимуществ, давайте сначала разберемся, для чего вообще может понадобиться объявление курсоров в пакете? Почему не объявить явный курсор в той программе, в которой он используется — в процедуре, функции или анонимном блоке?

Ответ прост и убедителен. Определяя курсор в пакете, можно многократно использовать заданный в нем запрос, не повторяя один и тот же код в разных местах приложения. Реализация запроса в одном месте упрощает его доработку и сопровождение кода. Некоторая экономия времени достигается за счет сокращения количества обрабатываемых запросов.

Также стоит рассмотреть возможность создания функции, возвращающей курсорную переменную на базе `REF CURSOR`. Вызывающая программа осуществляет выборку строк через курсорную переменную. За дополнительной информацией обращайтесь к разделу «Курсорные переменные и `REF CURSOR`».



Объявляя курсоры в пакетах для повторного использования, следует учитывать одно важное обстоятельство. Все структуры данных, в том числе и курсоры, объявляемые на «уровне пакета» (не внутри конкретной функции или процедуры), сохраняют свои значения на протяжении всего сеанса. Это означает, что пакетный курсор будет оставаться открытым до тех пор, пока вы явно не закроете его, или до завершения сеанса. Курсоры, объявленные в локальных блоках, автоматически закрываются при завершении этих блоков.

А теперь давайте разберемся с секцией `RETURN`. У объявления курсора в пакете имеется одна интересная особенность: заголовок курсора может быть отделен от его тела. Такой заголовок, больше напоминающий заголовок функции, содержит информацию, которая необходима программисту для работы: имя курсора, его параметры и тип возвращаемых данных. Телом курсора служит команда `SELECT`. Этот прием продемонстрирован в новой версии объявления курсора `books_cur` в пакете `book_info`:

```
PACKAGE book_info
IS
    CURSOR books_cur (title_filter_in IN books.title%TYPE)
        RETURN books%ROWTYPE;
END;

PACKAGE BODY book_info
IS
    CURSOR books_cur (title_filter_in IN books.title%TYPE)
        RETURN books%ROWTYPE
    IS
        SELECT *
          FROM books
         WHERE title LIKE title_filter_in;
END;
```

Все символы до ключевого слова `IS` образуют спецификацию, а после `IS` следует тело курсора. Разделение объявления курсора может служить двум целям.

- **Соккрытие информации.** Курсор в пакете представляет собой «черный ящик». Это удобно для программистов, потому что им не нужно ни писать, ни даже видеть команду `SELECT`. Достаточно знать, какие записи возвращает этот курсор, в каком порядке и какие столбцы они содержат. Программист, работающий с пакетом, использует курсор как любой другой готовый элемент.
- **Минимум перекомпиляции.** Если скрыть определение запроса в теле пакета, то изменения в команду `SELECT` можно будет вносить, не меняя заголовок курсора в спецификации пакета. Это позволяет совершенствовать, исправлять и повторно компилировать код без перекомпиляции спецификации пакета, благодаря чему зависящие от этого пакета программы не будут помечены как недействительные и их также не нужно будет перекомпилировать.

Открытие явного курсора

Использование курсора начинается с его определения в разделе объявлений. Далее объявленный курсор необходимо открыть. Синтаксис оператора `OPEN` очень прост:

```
OPEN имя_курсора [ ( аргумент [, аргумент ...] ) ];
```

Здесь *имя_курсора* — это имя объявленного ранее курсора, а *аргумент* — значение, передаваемое курсору, если он объявлен со списком параметров.



Oracle также поддерживает синтаксис `FOR` при открытии курсора, который используется как для курсорных переменных (см. раздел «Курсорные переменные и `REF CURSOR`»), так и для встроенного динамического SQL (см. главу 16).

Открывая курсор, PL/SQL выполняет содержащийся в нем запрос. Кроме того, он идентифицирует активный набор данных — строки всех участвующих в запросе таблиц, соответствующие критерию `WHERE` и условию объединения. Команда `OPEN` не извлекает данные — это задача команды `FETCH`.

Независимо от того, когда будет выполнена первая выборка данных, реализованная в Oracle модель целостности данных гарантирует, что все операции выборки будут возвращать данные в состоянии на момент открытия курсора. Иными словами, от открытия и до закрытия курсора при выборке из него данных полностью игнорируются выполняемые за это время операции вставки, обновления и удаления.

Более того, если команда `SELECT` содержит секцию `FOR UPDATE`, все идентифицируемые курсором строки блокируются при его открытии. (Данная возможность подробнее описана далее в этой главе, в разделе «Команда `SELECT...FOR UPDATE`».)

При попытке открыть уже открытый курсор PL/SQL выдаст следующее сообщение об ошибке:

```
ORA-06511: PL/SQL: cursor already open
```

Поэтому перед открытием курсора следует проверить его состояние по значению атрибута `%ISOPEN`:

```
IF NOT company_cur%ISOPEN  
THEN  
    OPEN company_cur;  
END IF;
```

Атрибуты явных курсоров описываются ниже, в посвященном им разделе.



Если в программе выполняется цикл `FOR` с использованием курсора, этот курсор не нуждается в явном открытии (выборке данных, закрытии). Ядро PL/SQL делает это автоматически.

Выборка данных из явного курсора

Команда `SELECT` создает *виртуальную таблицу* — набор строк, определяемых условием `WHERE` со столбцами, определяемыми списком столбцов `SELECT`. Таким образом, курсор представляет эту таблицу в программе PL/SQL. Основным назначением курсора в программах PL/SQL является выборка строк для обработки. Выборка строк курсора выполняется командой `FETCH`:

```
FETCH имя_курсора INTO запись_или_список_переменных;
```

Здесь *имя_курсора* — имя курсора, из которого выбирается запись, а *запись_или_список_переменных* — структуры данных PL/SQL, в которые копируется следующая строка активного набора записей. Данные могут помещаться в запись PL/SQL (объявленную с атрибутом `%ROWTYPE` или объявлением `TYPE`) или в переменные (переменные PL/SQL или переменные привязки — как, например, в элементы Oracle Forms).

Примеры явных курсоров

Следующие примеры демонстрируют разные способы выборки данных.

- Выборка данных из курсора в запись PL/SQL:

```
DECLARE  
    CURSOR company_cur is SELECT ...;  
    company_rec company_cur%ROWTYPE;
```

продолжение ➤

```
BEGIN
  OPEN company_cur;
  FETCH company_cur INTO company_rec;
```

- Выборка данных из курсора в переменную:

```
FETCH new_balance_cur INTO new_balance_dollars;
```
- Выборка данных из курсора в строку таблицы PL/SQL, переменную и переменную привязки Oracle Forms:

```
FETCH emp_name_cur INTO emp_name (1), hiredate, :dept.min_salary;
```



Данные, выбираемые из курсора, всегда следует помещать в запись, объявленную на основе того же курсора с атрибутом %ROWTYPE; избегайте выборки в списки переменных. Выборка в запись делает код более компактным и гибким, позволяет изменять список выборки без изменения команды `FETCH`.

Выборка после обработки последней строки

Открыв курсор, вы по очереди выбираете из него строки, пока они все не будут исчерпаны. Однако и после этого можно выполнять команду `FETCH`.

Как ни странно, в этом случае PL/SQL не иницирует исключение. Он просто ничего не делает. Поскольку выбирать больше нечего, значения переменных в секции `INTO` команды `FETCH` не изменяются. Иначе говоря, команда `FETCH` не устанавливает значения этих переменных равными `NULL`.

Следовательно, если вам понадобится узнать, успешно ли прошла выборка очередной строки, проверка переменных из списка `INTO` ничего не даст. Вместо этого следует проверить атрибут `%FOUND` или `%NOTFOUND` (см. далее раздел «Атрибуты явных курсоров»).

Псевдонимы столбцов явного курсора

Команда `SELECT` в объявлении курсора определяет список возвращаемых им столбцов. Наряду с именами столбцов таблиц этот список может содержать выражения, называемые *вычисляемыми*, или *виртуальными* столбцами.

Псевдоним (alias) столбца представляет собой альтернативное имя, указанное в команде `SELECT` для столбца или выражения. Задав подходящие псевдонимы в SQL*Plus, можно вывести результаты произвольного запроса в удобочитаемом виде. В подобных ситуациях псевдонимы не являются обязательными. С другой стороны, при использовании явных курсоров псевдонимы вычисляемых столбцов необходимы в следующих случаях:

- при выборке данных из курсора в запись, объявленную с атрибутом %ROWTYPE на основе того же курсора;
- когда в программе содержится ссылка на вычисляемый столбец.

Рассмотрим следующий запрос. Команда `SELECT` выбирает названия всех компаний, заказывавших товары в течение 2001 года, а также общую сумму заказов (предполагается, что для текущего экземпляра базы данных по умолчанию используется маска форматирования `DD-MON-YYYY`):

```
SELECT company_name, SUM (inv_amt)
  FROM company c, invoice i
 WHERE c.company_id = i.company_id
    AND i.invoice_date BETWEEN '01-JAN-2001' AND '31-DEC-2001';
```

При выполнении этой команды в SQL*Plus будет получен следующий результат:

| COMPANY_NAME | SUM (INV_AMT) |
|---------------------|---------------|
| ACME TURBO INC. | 1000 |
| WASHINGTON HAIR CO. | 25.20 |

Как видите, заголовок столбца SUM (INV_AMT) плохо подходит для отчета, но для простого просмотра данных он вполне годится. Теперь выполним тот же запрос в программе PL/SQL с использованием явного курсора и добавим псевдоним столбца:

```
DECLARE
    CURSOR comp_cur IS
        SELECT c.name, SUM (inv_amt) total_sales
          FROM company C, invoice I
         WHERE C.company_id = I.company_id
            AND I.invoice_date BETWEEN '01-JAN-2001' AND '31-DEC-2001';
    comp_rec comp_cur%ROWTYPE;
BEGIN
    OPEN comp_cur;
    FETCH comp_cur INTO comp_rec;
    ...
END;
```

Без псевдонима я не смогу сослаться на столбец в структуре записи `comp_rec`. При наличии псевдонима с вычисляемым столбцом можно работать точно так же, как с любым другим столбцом запроса:

```
IF comp_rec.total_sales > 5000
THEN
    DBMS_OUTPUT.PUT_LINE
        (' You have exceeded your credit limit of $5000 by ' ||
          TO_CHAR (comp_rec.total_sales - 5000, '$9999'));
END IF;
```

При выборке строки в запись, объявленную с атрибутом `%ROWTYPE`, доступ к вычисляемому столбцу можно будет получить только по имени — ведь структура записи определяется структурой самого курсора.

Заккрытие явного курсора

Когда-то в детстве нас учили прибирать за собой, и эта привычка осталась у нас (хотя и не у всех) на всю жизнь. Оказывается, это правило играет исключительно важную роль и в программировании, и особенно когда дело доходит до управления курсорами. Никогда не забывайте закрыть курсор, если он вам больше не нужен!

Синтаксис команды `CLOSE`:

```
CLOSE имя_курсора;
```

Ниже приводится несколько важных советов и соображений, связанных с закрытием явных курсоров.

- Если курсор объявлен и открыт в процедуре, не забудьте его закрыть после завершения работы с ним; в противном случае в вашем коде возникнет утечка памяти. Теоретически курсор (как и любая структура данных) должен автоматически закрываться и уничтожаться при выходе из области действия. Как правило, при выходе из процедуры, функции или анонимного блока PL/SQL действительно закрывает все открытые в нем курсоры. Но этот процесс связан с определенными затратами ресурсов, поэтому по соображениям эффективности PL/SQL иногда *откладывает* выявление и закрытие открытых курсоров. Курсоры типа `REF CURSOR` по определению не могут быть закрыты неявно. Единственное, в чем можно быть уверенным, так это в том, что по завершении работы «самого внешнего» блока PL/SQL, когда управление будет возвращено SQL или другой вызывающей программе, PL/SQL неявно закроет все открытые этим блоком или вложенными блоками курсоры, кроме `REF CURSOR`.



В статье «Cursor reuse in PL/SQL static SQL» из Oracle Technology Network приводится подробный анализ того, как и когда PL/SQL закрывает курсоры. Вложенные анонимные блоки — пример ситуации, в которой PL/SQL не осуществляет неявное закрытие курсоров. Интересная информация по этой теме приведена в статье Джонатана Генника «Does PL/SQL Implicitly Close Cursors?».

- Если курсор объявлен в пакете на уровне пакета и открыт в некотором блоке или программе, он останется открытым до тех пор, пока вы его явно не закроете, или до завершения сеанса. Поэтому, завершив работу с курсором пакетного уровня, его следует немедленно закрыть командой CLOSE (и кстати, то же самое следует делать в разделе исключений):

```
BEGIN
    OPEN my_package.my_cursor;

    ... Работаем с курсором

    CLOSE my_package.my_cursor;
EXCEPTION
    WHEN OTHERS
    THEN
        IF mypackage.my_cursor%ISOPEN THEN
            CLOSE my_package.my_cursor;
        END IF;
END;
```

- Курсор можно закрывать только в том случае, если ранее он был открыт; в противном случае будет инициировано исключение INVALID_CURSOR. Состояние курсора проверяется с помощью атрибута %ISOPEN:

```
IF company_cur%ISOPEN
THEN
    CLOSE company_cur;
END IF;
```

- Если в программе останется слишком много открытых курсоров, их количество может превысить значение параметра базы данных OPEN_CURSORS. Получив сообщение об ошибке, прежде всего убедитесь в том, что объявленные в пакетах курсоры закрываются после того, как надобность в них отпадет.

Атрибуты явных курсоров

Oracle поддерживает четыре атрибута (%FOUND, %NOTFOUND, %ISOPEN, %ROWCOUNTM) для получения информации о состоянии явного курсора. Ссылка на атрибут имеет следующий синтаксис:

курсor%атрибут

Здесь *курсor* — имя объявленного курсора.

Значения, возвращаемые атрибутами явных курсоров, приведены в табл. 15.3.

Таблица 15.3. Атрибуты явных курсоров

| Имя | Что возвращает |
|-----------------|--|
| курсor%FOUND | TRUE, если строка выбрана успешно |
| курсor%NOTFOUND | TRUE, если не была выбрана ни одна строка |
| курсor%ROWCOUNT | Количество строк, выбранных из заданного курсора до настоящего момента |
| курсor%ISOPEN | TRUE, если заданный курсор открыт |

Значения атрибутов курсоров до и после выполнения различных операций с ними указаны в табл. 15.4.

Работая с атрибутами явных курсоров, необходимо учитывать следующее:

- При попытке обратиться к атрибуту %FOUND, %NOTFOUND или %ROWCOUNT до открытия курсора или после его закрытия Oracle инициирует исключение *INVALID_CURSOR (ORA-01001)*.
- Если после первого выполнения команды FETCH результирующий набор строк окажется пустым, атрибуты курсора возвращают следующие значения: %FOUND = FALSE, %NOTFOUND = TRUE и %ROWCOUNT = 0.
- При использовании BULK COLLECT атрибут %ROWCOUNT возвращает количество строк, извлеченных в заданные коллекции. За подробностями обращайтесь к главе 21.

Таблица 15.4. Значения атрибутов курсоров

| Операция | %FOUND | %NOTFOUND | %ISOPEN | %ROWCOUNT |
|-----------------------------------|-------------------------|-------------------------|---------|-------------------------|
| До OPEN | Исключение ORA-01001 | Исключение ORA-01001 | FALSE | Исключение ORA-01001 |
| После OPEN | NULL | NULL | TRUE | 0 |
| До первой выборки FETCH | NULL | NULL | TRUE | 0 |
| После первой выборки FETCH | TRUE | FALSE | TRUE | 1 |
| Перед последующими FETCH | TRUE | FALSE | TRUE | 1 |
| После последующих FETCH | TRUE | FALSE | TRUE | Зависит от данных |
| Перед последней выборкой FETCH | TRUE | FALSE | TRUE | Зависит от данных |
| После последней выборки FETCH | TRUE | FALSE | TRUE | Зависит от данных |
| Перед CLOSE | FALSE | TRUE | TRUE | Зависит от данных |
| После CLOSE | Исключение | Исключение | FALSE | Исключение |

Использование всех этих атрибутов продемонстрировано в следующем примере:

```

PACKAGE bookinfo_pkg
IS
    CURSOR bard_cur
    IS SELECT title, date_published
       FROM books
       WHERE UPPER(author) LIKE 'SHAKESPEARE%';
END bookinfo_pkg;

DECLARE
    bard_rec    bookinfo_pkg.bard_cur%ROWTYPE;
BEGIN
    /* Проверяем, не открыт ли уже курсор.
       Это возможно, поскольку курсор определен в пакете.
       Если курсор открыт, закрываем его и открываем повторно,
       чтобы получить "свежий" результирующий набор.
    */
    IF bookinfo_pkg.bard_cur%ISOPEN
    THEN
        CLOSE bookinfo_pkg.bard_cur;
    END IF;

    OPEN bookinfo_pkg.bard_cur;

    -- По очереди выбираем строки. Перебор останавливается
    -- после вывода первых пяти произведений Шекспира
    -- или когда будут исчерпаны все строки.

```

продолжение ⇨

```

LOOP
    FETCH bookinfo_pkg.bard_cur INTO bard_rec;
    EXIT WHEN bookinfo_pkg.bard_cur%NOTFOUND
        OR bookinfo_pkg.bard_cur%ROWCOUNT > 5;
    DBMS_OUTPUT.put_line (
        bookinfo_pkg.bard_cur%ROWCOUNT
        || ' '
        || bard_rec.title
        || ', издана в '
        || TO_CHAR (bard_rec.date_published, 'YYYY')
    );
END LOOP;
CLOSE bookinfo_pkg.bard_cur;
END;
```

Параметры курсора

В этой книге уже неоднократно приводились примеры использования параметров процедур и функций. Параметры — это средство передачи информации в программный модуль и из него. При правильном использовании они делают модули более полезными и гибкими.

PL/SQL позволяет передавать параметры курсорам. Они выполняют те же функции, что и параметры программных модулей, а также несколько дополнительных.

- **Расширение возможности многократного использования курсоров.** Вместо того чтобы жестко кодировать в предложении `WHERE` значения, определяющие условия отбора данных, можно использовать параметры для передачи в это предложение новых значений при каждом открытии курсора.
- **Решение проблем, связанных с областью действия курсоров.** Если вместо жестко закодированных значений в запросе используются параметры, результирующий набор строк курсора не привязан к конкретной переменной программы или блока. Если в программе имеются вложенные блоки, курсор можно определить на верхнем уровне и использовать его во вложенных блоках с объявленными в них переменными.

Количество параметров курсора не ограничено. При вызове `OPEN` для курсора должны быть заданы значения всех параметров (кроме параметров, для которых определены значения по умолчанию).

В каких случаях курсору требуются параметры? Общее правило здесь то же, что и для процедур и функций: если предполагается, что курсор будет использоваться в разных местах и с разными значениями в секции `WHERE`, для него следует определить параметр.

Давайте сравним курсоры с параметром и без. Пример курсора без параметров:

```

CURSOR joke_cur IS
    SELECT name, category, last_used_date
    FROM jokes;
```

В результирующий набор курсора включаются все записи таблицы `joke`. Если же нам нужно только некоторое подмножество строк, в запрос включается секция `WHERE`:

```

CURSOR joke_cur IS
    SELECT name, category, last_used_date
    FROM jokes
    WHERE category = 'HUSBAND';
```

Для выполнения этой задачи мы не стали использовать параметры, да они и не нужны. В данном случае курсор возвращает все строки, относящиеся к конкретной категории. Но как быть, если при каждом обращении к этому курсору категория изменяется?

Курсоры с параметрами

Конечно, мы не станем определять отдельный курсор для каждой категории — это совершенно не согласуется с принципом разработки приложений, управляемых данными. Нам нужен всего один курсор, но такой, для которого можно было бы менять категорию — и он все равно возвращал бы требуемую информацию. И лучшим (хотя и не единственным) решением этой задачи является определение параметризованного курсора:

```
PROCEDURE explain_joke (main_category_in IN joke_category.category_id%TYPE)
IS
    /*
    || Курсор со списком параметров, состоящим
    || из единственного строкового параметра.
    */
    CURSOR joke_cur (category_in IN VARCHAR2)
    IS
        SELECT name, category, last_used_date
        FROM joke
        WHERE category = UPPER (category_in);
    joke_rec joke_cur%ROWTYPE;

BEGIN
    /* Теперь при открытии курсора ему передается аргумент */
    OPEN joke_cur (main_category_in);
    FETCH joke_cur INTO joke_rec;
```

Между именем курсора и ключевым словом **IS** теперь содержится список параметров. Жестко закодированное значение **HUSBAND** в предложении **WHERE** заменено ссылкой на параметр **UPPER (category_in)**. При открытии курсора можно будет задать значение **HUSBAND**, **husband** или **hUsbAnD** — курсор все равно будет работать. Название категории, для которой курсор должен вернуть строки таблицы **joke**, задается в операторе **OPEN** (в скобках) в виде литерала, константы или выражения. В момент открытия курсора производится разбор команды **SELECT**, а параметр связывается со значением. Затем определяется результирующий набор строк — и курсор готов к выборке.

Открытие курсора с параметрами

Новый курсор можно открывать с указанием любой категории:

```
OPEN joke_cur ( Jokes_pkg.category);
OPEN joke_cur ('husband');
OPEN joke_cur ('politician');
OPEN joke_cur ( Jokes_pkg.relation || '-IN-LAW');
```

Параметры курсора чаще всего используются в условии **WHERE**, но ссылаться на них можно и в других местах команды **SELECT**:

```
DECLARE
    CURSOR joke_cur (category_in IN ARCHAR2)
    IS
        SELECT name, category_in, last_used_date
        FROM joke
        WHERE category = UPPER (category_in);
```

Вместо того чтобы считывать категорию из таблицы, мы просто подставляем параметр **category_in** в список выборки. Результат остается прежним, потому что условие **WHERE** ограничивает категорию выборки значением параметра.

Область действия параметра курсора

Область действия параметра курсора ограничивается этим курсором. На параметр курсора нельзя ссылаться за пределами команды **SELECT**, связанной с курсором. Приведенный ниже фрагмент PL/SQL не компилируется, потому что идентификатор **program_name** не

является локальной переменной в блоке. Это формальный параметр курсора, который определен только внутри курсора:

```
DECLARE
  CURSOR scariness_cur (program_name VARCHAR2)
  IS
    SELECT SUM (scary_level) total_scary_level
    FROM tales_from_the_crypt
    WHERE prog_name = program_name;
BEGIN
  program_name := 'THE BREATHING MUMMY'; /* Недопустимая ссылка */
  OPEN scariness_cur (program_name);
  ...
  CLOSE scariness_cur;
END;
```

Режимы параметра курсора

Синтаксис параметров курсоров очень похож на синтаксис процедур и функций — за исключением того, что параметры курсоров могут быть только параметрами IN. Для параметров курсоров нельзя задавать режимы OUT или IN OUT. Эти режимы позволяют передавать и возвращать значения из процедур, что не имеет смысла для курсора. Существует только один способ получения информации от курсора: выборка записи и копирование значений из списка столбцов в секции INTO (за дополнительной информацией о режимах параметров обращайтесь к главе 17).

Значения параметров по умолчанию

Параметрам курсоров могут присваиваться значения по умолчанию. Пример курсора со значением параметра по умолчанию:

```
CURSOR emp_cur (emp_id_in NUMBER := 0)
IS
  SELECT employee_id, emp_name
  FROM employee
  WHERE employee_id = emp_id_in;
```

Поскольку для параметра `emp_id_in` определено значение по умолчанию, в команде `FETCH` его значение можно не указывать. В этом случае курсор вернет информацию о сотруднике с кодом 0.

Команда SELECT...FOR UPDATE

При выполнении команды `SELECT` для выборки строк из базы данных эти строки не блокируются. Обычно это чрезвычайно удобно, потому что количество записей, заблокированных в конкретный момент времени, сводится к минимуму: блокируются только те строки, которые уже изменены, но еще не зафиксированы приложением. Но даже эти строки доступны для чтения в том состоянии, в котором они находились до внесения изменений.

Однако в некоторых случаях требуется заблокировать набор строк еще до того, как вы приступите к их изменению в программе. Это можно сделать в команде `SELECT` с помощью секции `FOR UPDATE`.

При выполнении команды `SELECT...FOR UPDATE` Oracle автоматически блокирует все строки, определяемые командой `SELECT`. Никто другой не сможет изменять эти строки до тех пор, пока не будет выполнена команда `ROLLBACK` или `COMMIT`, хотя другие сеансы по-прежнему могут читать из них данные.

Рассмотрим пару примеров, демонстрирующих использование предложения `FOR UPDATE` в курсорах:

```
CURSOR toys_cur IS
  SELECT name, manufacturer, preference_level, sell_at_yardsale_flag
  FROM my_sons_collection
  WHERE hours_used = 0
  FOR UPDATE;

CURSOR fall_jobs_cur IS
  SELECT task, expected_hours, tools_required, do_it_yourself_flag
  FROM winterize
  WHERE year_of_task = TO_CHAR (SYSDATE, 'YYYY')
  FOR UPDATE OF task;
```

В первом курсоре используется секция FOR UPDATE без параметров, а во втором — с заданным именем столбца.

Секция FOR UPDATE может применяться в командах SELECT, выбирающих данные из нескольких таблиц. Если при этом в указанном предложении имеется список OF, блокируются строки только тех таблиц, строки которых указываются в этом списке. Если в команде SELECT присутствует секция FOR UPDATE без списка OF, Oracle блокирует все отобранные запросом строки всех таблиц из секции FROM.

Более того, команда SELECT...FOR UPDATE вообще не требует последующего выполнения команд DELETE или UPDATE — она просто сообщает Oracle, что вы собираетесь это сделать (и не позволяет это делать другим).

Наконец, секцию FOR UPDATE можно дополнить ключевым словом NOWAIT. Оно означает, что если таблица заблокирована другим пользователем, Oracle не следует ждать ее освобождения. В этом случае управление будет сразу возвращено программе, чтобы она могла заняться другой работой или просто подождать некоторое время перед повторной попыткой. Также к команде можно присоединить ключевое слово WAIT с указанием максимальной продолжительности ожидания блокировки в секундах. Если ни одно из этих ключевых слов не указано, сеанс блокируется до тех пор, пока таблица не освободится. Причем тайм-аут в данном случае предусмотрен только для удаленных таблиц — он определяется значением инициализационного параметра DISTRIBUTED_LOCK_TIMEOUT.

Снятие блокировок командой COMMIT

Как только для курсора с секцией FOR UPDATE будет выполнена команда OPEN, все строки его результирующего набора блокируются и остаются заблокированными до тех пор, пока внесенные в сеанс изменения не будут закреплены командой COMMIT или отменены командой ROLLBACK. В любом случае заблокированные строки при этом освобождаются; таким образом, после выполнения COMMIT или ROLLBACK текущая позиция в курсоре теряется, и вы не сможете выполнить следующую выборку командой FETCH.

Рассмотрим пример:

```
DECLARE
  /* Подготовка к зиме */
  CURSOR fall_jobs_cur
  IS
    SELECT task, expected_hours, tools_required, do_it_yourself_flag
    FROM winterize
    WHERE year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'))
      AND completed_flag = 'NOTYET';
BEGIN
  /* Для каждой строки, выбранной курсором... */
  FOR job_rec IN fall_jobs_cur
  LOOP
    IF job_rec.do_it_yourself_flag = 'YOUcandoIT'
```

продолжение ➤

```

THEN
  /*
  || Найдено очередное задание, фиксируем изменения.
  */
  UPDATE winterize SET responsible = 'STEVEN'
  WHERE task = job_rec.task
  AND year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));
  COMMIT;
END IF;
END LOOP;
END;
```

Предположим, этот цикл нашел первое задание с пометкой **YOUcandoIT**. Ответственным за его выполнение назначается **STEVEN**, но при попытке выборки следующей строки происходит исключение:

ORA-01002: fetch out of sequence

Если при извлечении данных из курсора **SELECT...FOR UPDATE** необходимо производить закрепление или откат изменений, придется включить в программу код, прекращающий дальнейшую выборку (например, команда **EXIT**, выполняющая выход из цикла).

Предложение WHERE CURRENT OF

PL/SQL позволяет использовать в командах **UPDATE** и **DELETE** специальную конструкцию **WHERE CURRENT OF**, облегчающую процесс изменения последней выбранной из курсора строки данных.

Обновление столбцов последней выбранной строки можно выполнить следующей командой:

```

UPDATE имя_таблицы
  SET предложение_set
  WHERE CURRENT OF имя_курсора;
```

Аналогичным образом производится удаление последней выбранной строки:

```

DELETE
  FROM имя_таблицы
  WHERE CURRENT OF имя_курсора;
```

Обратите внимание: в секции **WHERE CURRENT OF** указывается курсор, а не запись, в которую была помещена очередная строка.

Главное преимущество использования секции **WHERE CURRENT OF** при модификации или удалении последней извлеченной строки из курсора заключается в том, что один критерий поиска строки не нужно задавать в двух (и более) местах программы. Не будь его, пришлось бы ввести секцию **WHERE** в определение курсора, а затем повторить его в соответствующих командах **UPDATE** и **DELETE**. Если бы в будущем структура таблицы изменилась способом, влияющим на формирование первичного ключа, нам пришлось бы вносить изменения во всех командах **SQL**, в которых оно используется. С другой стороны, с **WHERE CURRENT OF** изменяется только секция **WHERE** команды **SELECT**.

На первый взгляд этот момент кажется второстепенным, но это одна из многих областей кода, которые могут использовать неочевидные возможности PL/SQL для сведения к минимуму избыточности в коде. Использование **WHERE CURRENT OF**, атрибутов **%TYPE** и **%ROWTYPE**, циклов **FOR** с курсором, локальной модуляризации и других языковых конструкций PL/SQL может существенно упростить сопровождение приложений Oracle.

Давайте посмотрим, как с помощью синтаксиса **WHERE CURRENT OF** усовершенствовать пример из предыдущего раздела. В цикле **FOR** нужно обновить строку, только что выбранную из курсора. Попробуем сделать это с помощью команды **UPDATE**, в которой задано то же

условие, что в команде SELECT курсора (первичный ключ таблицы составляют значения столбцов task и year):

```
WHERE task = job_rec.task  
AND year = TO_CHAR (SYSDATE, 'YYYY');
```

Как было указано ранее, это неверный подход: одна и та же логика программируется в двух местах, и при внесении каких-либо изменений придется следить за синхронизацией этого кода. Было бы намного проще и естественнее, если бы PL/SQL предоставлял такие операции, как удаление только что выбранной строки и обновление ее столбцов.

Именно это и делает секция WHERE CURRENT OF! В новой версии предыдущего примера мы воспользуемся им, а заодно заменим цикл FOR простым циклом с условным выходом (в цикле FOR так поступать можно, но не рекомендуется):

```
DECLARE  
    CURSOR fall_jobs_cur IS SELECT ... то же, что в предыдущем примере ... ;  
    job_rec fall_jobs_cur%ROWTYPE;  
BEGIN  
    OPEN fall_jobs_cur;  
    LOOP  
        FETCH fall_jobs_cur INTO job_rec;  
  
        EXIT WHEN fall_jobs_cur%NOTFOUND;  
  
        IF job_rec.do_it_yourself_flag = 'YOUcandoIT'  
        THEN  
            UPDATE winterize SET responsible = 'STEVEN'  
            WHERE CURRENT OF fall_jobs_cur;  
            COMMIT;  
            EXIT;  
        END IF;  
    END LOOP;  
    CLOSE fall_jobs_cur;  
END;
```

Курсорные переменные и REF CURSOR

Курсорная переменная ссылается на курсор. В отличие от явного курсора, имя которого в PL/SQL используется как идентификатор рабочей области результирующего набора строк, курсорная переменная содержит ссылку на эту рабочую область. Явные и неявные курсоры имеют статическую природу, поскольку они жестко привязаны к конкретным запросам. С помощью же курсорной переменной можно выполнить любой запрос и даже несколько разных запросов в одной программе.

Важнейшим преимуществом курсорных переменных является то, что они предоставляют механизм передачи результатов запроса (выбранных из строк курсора) между разными программами PL/SQL, в том числе между клиентскими и серверными программами. До выхода PL/SQL Release 2.3 приходилось выбирать из курсора все данные, сохраняя их в переменных PL/SQL (например, в коллекции) и передавать эти данные в аргументах. А курсорная переменная позволяет передать другой программе ссылку на объект курсора, чтобы та могла работать с его данными. Это упрощает программный код и повышает его эффективность.

Кроме того, курсоры могут совместно использоваться несколькими программами. Например, в архитектуре «клиент-сервер» клиентская программа может открыть курсор и начать выборку из него данных, а затем передать указывающую на него переменную в качестве аргумента хранимой процедуре на сервере. Эта процедура продолжит выборку, а некоторое время спустя снова передаст управление клиентской программе, которая

закроет курсор. Так же могут взаимодействовать и две разные хранимые программы из одного или разных экземпляров Oracle.

Описанный процесс схематически показан на рис. 15.2. Он демонстрирует интересные новые возможности программ PL/SQL — совместное использование данных и управление курсорами.

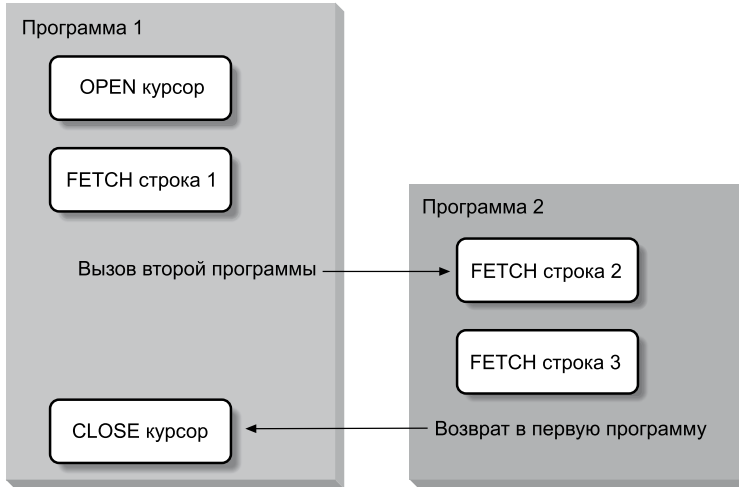


Рис. 15.2. Передача ссылки на курсор между программами

Когда используются курсорные переменные?

Ниже перечислены основные области применения курсорных переменных.

- Курсорные переменные могут связываться с разными запросами в разные моменты выполнения вашей программы. Иначе говоря, одна курсорная переменная может использоваться для выборки данных из разных результирующих наборов.
- Курсорные переменные могут передаваться в аргументах процедур или функций. По сути, передавая ссылку на результирующий набор курсора, вы организуете совместное использование его результатов.
- Полноценное использование функциональности статических курсоров PL/SQL. Вы можете применять OPEN, CLOSE и FETCH к своим курсорным переменным в программах PL/SQL, а также обращаться к стандартным атрибутам курсоров — %ISOPEN, %FOUND, %NOTFOUND и %ROWCOUNT.
- Присваивание содержимого одного курсора (и его результирующего набора) другой курсорной переменной. Курсорная переменная, как и любая другая переменная, может использоваться в операциях присваивания. Однако, как будет показано позднее в этой главе, на применение таких переменных устанавливаются определенные ограничения.

Сходство со статическими курсорами

При проектировании курсорных переменных одно из важнейших требований заключалось в том, что семантика управления объектами курсоров по возможности должна совпадать с семантикой управления статическими курсорами. Хотя объявления курсорных переменных и синтаксис их открытия были усовершенствованы, следующие операции с курсорными переменными не отличаются от операций со статическими курсорами:

○ Команда CLOSE

В следующем примере объявляется тип REF CURSOR и курсорная переменная для этого типа. Затем курсорная переменная закрывается с использованием того же синтаксиса, что и для статических курсоров:

```
DECLARE
    TYPE var_cur_type IS REF CURSOR;
    var_cur var_cur_type;
BEGIN
    OPEN var_cur FOR ...
    ...
    CLOSE var_cur;
END;
```

○ Атрибуты курсоров

Разработчик может использовать любые из четырех атрибутов курсоров с точно таким же синтаксисом, как и у статических курсоров. По правилам, управляющим их использованием, и значениям, возвращаемым атрибутами, они ничем не отличаются от явных курсоров. Так, для объявления курсорной переменной из предыдущего примера могут использоваться следующие атрибуты:

```
var_cur%ISOPEN
var_cur%FOUND
var_cur%NOTFOUND
var_cur%ROWCOUNT
```

○ Выборка данных из курсорной переменной

При выборке данных из курсорной переменной в локальные структуры данных PL/SQL может использоваться уже знакомый синтаксис FETCH. Однако PL/SQL устанавливает дополнительные правила, которые гарантируют, что структуры данных строки курсорной переменной (набор значений, возвращаемых объектом курсора) соответствуют структурам данных справа от ключевого слова INTO. Эти правила рассматриваются в разделе «Правила использования курсорных переменных» этой главы.

Так как синтаксис этих аспектов курсорных переменных не отличается от синтаксиса уже знакомых нам явных курсоров, в следующих разделах основное внимание будет уделяться уникальным особенностям курсорных переменных.

Объявление типов REF CURSOR

По аналогии с таблицами PL/SQL или записями типа, определяемого программистом, курсорная переменная объявляется в два этапа.

1. Определение типа курсора.
2. Объявление фактической переменной на базе этого типа.

Синтаксис объявления типа курсора:

```
TYPE имя_типа_курсора IS REF CURSOR [ RETURN возвращаемый_тип];
```

Здесь *имя_курсора* — имя типа курсора, а *возвращаемый_тип* — спецификация возвращаемых данных курсора. Это может быть любая структура данных, использование которой допускается в секции RETURN, определенная с помощью атрибута %ROWTYPE или путем ссылки на ранее объявленный тип записи.

Обратите внимание, что секция RETURN в объявлениях типа REF CURSOR не обязательна, поэтому допустимы оба следующих объявления:

```
TYPE company_curtype IS REF CURSOR RETURN company%ROWTYPE;
TYPE generic_curtype IS REF CURSOR;
```

Первая форма REF CURSOR называется *сильнотипизированной*, поскольку тип структуры, возвращаемой курсорной переменной, задается в момент объявления (непосредственно

или путем привязки к типу строки таблицы). Курсорная переменная, объявленная на основе данного типа, может использоваться только для выборки в структуры данных, соответствующие типу записи. Преимущество сильной типизации заключается в том, что компилятор может сразу определить, соответствует ли тип заданной в нем структуры данных спецификации RETURN в определении REF CURSOR.

Вторая форма (без предложения RETURN) называется *слаботипизированной*. Тип возвращаемой структуры данных для нее не задается. Курсорная переменная, объявленная на основе такого типа, обладает большей гибкостью, поскольку для нее можно задавать любые запросы с любой структурой возвращаемых данных, причем с помощью одной и той же переменной можно поочередно выполнить несколько разных запросов с результатами разных типов.

В Oracle9i появился предопределенный слабый тип REF CURSOR с именем SYS_REFCURSOR. Теперь программисту не нужно определять собственный слабый тип — достаточно использовать стандартный тип Oracle:

```
DECLARE
    my_cursor SYS_REFCURSOR;
```

Объявление курсорной переменной

Синтаксис объявления курсорной переменной таков:

имя_курсорной_переменной *имя_типа_курсора*

Здесь *имя_курсорной_переменной* — имя объявляемой переменной, а *имя_типа_курсора* — имя типа курсора, объявленного ранее с помощью команды TYPE.

Пример создания курсорной переменной:

```
DECLARE
    /* Создание типа курсора для информации о спортивных автомобилях. */
    TYPE sports_car_cur_type IS REF CURSOR RETURN car%ROWTYPE;
    /* Create a cursor variable for sports cars. */
    sports_car_cur sports_car_cur_type;
BEGIN
    ...
END;
```

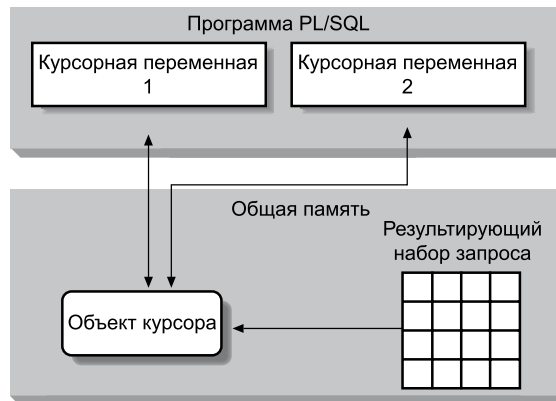


Рис. 15.3. Курсоры и курсорные переменные

Важно понимать различие между операцией объявления курсорной переменной и операцией создания реального объекта курсора, то есть результирующего набора строк, определяемого командой SQL курсора. Как известно, константа является значением,

а переменная — указателем на значение; аналогичным образом статический набор является набором данных, а курсорная переменная — указателем на этот набор. Различие продемонстрировано на рис. 15.3. Обратите внимание на то, что две разные переменные в разных программах ссылаются на один и тот же объект курсора.

При объявлении курсорной переменной объект курсора не создается. Для создания такого, а также для помещения ссылки на него в переменную нужно выполнить команду `OPEN FOR`.

Открытие курсорной переменной

Значение (объект курсора) присваивается курсорной переменной при открытии курсора. Таким образом, синтаксис традиционной команды `OPEN` позволяет использовать после секции `FOR` команду `SELECT`:

```
OPEN имя_курсора FOR команда_select;
```

Здесь *имя_курсора* — имя курсора или курсорной переменной, а *команда_select* — команда SQL `SELECT`, используемая для формирования набора строк курсора.

Для сильнотипизированной курсорной переменной количество и типы данных столбцов, указанных в команде `SELECT`, должны быть совместимыми со структурой, заданной в секции `RETURN` команды `TYPE . . . REF CURSOR`. Пример совместимых типов показан на рис. 15.4. Подробнее о совместимости возвращаемых курсорами структур рассказывается далее, в разделе «Правила использования курсорных переменных».

```
DECLARE
  TYPE emp_curtype IS
    REF CURSOR RETURN emp%ROWTYPE;
  emp_curvar emp_curtype;
BEGIN
  OPEN emp_curvar FOR
    SELECT * FROM emp;
END;
```

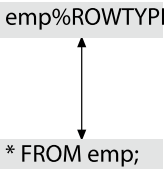


Рис. 15.4. Совместимость типа данных REF CURSOR и типа данных значений, возвращаемых командой `SELECT`

Если курсорная переменная объявлена со слабой формой `REF CURSOR`, ее можно открывать с любым запросом, независимо от структуры возвращаемых им данных. В следующем примере курсорная переменная трижды открывается с тремя разными запросами (а последняя команда `OPEN` вообще никак не связана с таблицей `employee!`):

```
DECLARE
  TYPE emp_curtype IS REF CURSOR;
  emp_curvar emp_curtype;
BEGIN
  OPEN emp_curvar FOR SELECT * FROM employees;
  OPEN emp_curvar FOR SELECT employee_id FROM employees;
  OPEN emp_curvar FOR SELECT company_id, name FROM company;
END;
```

Если курсорной переменной еще не присвоен никакой объект курсора, команда `OPEN FOR` неявно создает его. Если же переменная уже указывает на объект курсора, команда `OPEN FOR` не создает новый курсор, а ассоциирует с существующим новый запрос. Таким образом, объект курсора является структурой, отдельной и от курсорной переменной, и от запроса.



Если с курсорной переменной, которая ранее использовалась с командой OPEN FOR, связывается новый результирующий набор и эта курсорная переменная не была явно закрыта, курсор остается открытым. Всегда явно закрывайте курсорные переменные перед тем, как связывать их с другим результирующим набором.

Выборка данных из курсорной переменной

Как уже упоминалось, синтаксис команды FETCH для курсорной переменной не отличается от синтаксиса статического курсора:

```
FETCH имя_курсорной_переменной INTO имя_записи;
FETCH имя_курсорной_переменной INTO имя_переменной, имя_переменной...;
```

Если курсорная переменная объявляется в сильнотипизированной форме, компилятор PL/SQL проверяет совместимость типов структур в предложении INTO со структурой запроса, связанного с курсорной переменной.

Если курсорная переменная объявляется в слаботипизированной форме, компилятор не сможет выполнить подобную проверку. Данные из такой курсорной переменной могут извлекаться в любые структуры данных, поскольку компилятор не знает, какой объект курсора (и какая команда SQL) будет ей присвоен на момент выборки.

Поэтому проверка совместимости производится во время выполнения программы, непосредственно в момент выборки данных. И если окажется, что возвращенные запросом данные не соответствуют структурам, указанным в условии INTO, исполняющее ядро PL/SQL сгенерирует исключение ROWTYPE_MISMATCH. Учтите, что при необходимости и наличии возможности PL/SQL выполняет неявное преобразование данных.

Обработка исключения ROWTYPE_MISMATCH

Вы можете перехватить исключение ROWTYPE_MISMATCH, а затем попытаться выполнить выборку из курсорной переменной с другой секцией INTO. Но несмотря на выполнение второй команды FETCH в программе, вы все равно получите первую строку результирующего набора запроса объекта курсора. Эта функциональность удобна для слабых типов REF CURSOR, которые легко определяются с использованием предопределенного типа SYS_REFCURSOR.

В следующем примере в таблицах централизованной базы данных хранится информация о разных видах недвижимости: одна таблица для домов, другая для коммерческих строений и т. д. Также существует одна центральная таблица с адресами и типами зданий (жилой дом, коммерческое сооружение и т. д.). Я использую одну процедуру для открытия слабой переменной REF CURSOR для таблицы, соответствующей адресу объекта. Затем каждое бюро недвижимости вызывает эту процедуру для перебора соответствующих строений. Общая последовательность действий выглядит так:

1. Создание процедуры. Обратите внимание: параметр курсорной переменной определяется в режиме IN OUT:

```
/* Файл в Сети: rowtype_mismatch.sql */
PROCEDURE open_site_list
  (address_in IN VARCHAR2,
   site_cur_inout IN OUT SYS_REFCURSOR)
IS
  home_type CONSTANT PLS_INTEGER := 1;
  commercial_type CONSTANT PLS_INTEGER := 2;

  /* Статический курсор для получения типа строения. */
  CURSOR site_type_cur IS
```

```

        SELECT site_type FROM property_master
        WHERE address = address_in;
        site_type_rec site_type_cur%ROWTYPE;

BEGIN
    /* Получение типа строения для данного адреса. */
    OPEN site_type_cur;
    FETCH site_type_cur INTO site_type_rec;
    CLOSE site_type_cur;

    /* Использование типа для выборки из правильной таблицы.*/
    IF site_type_rec.site_type = home_type
    THEN
        /* использование таблицы жилых домов. */
        OPEN site_cur_inout FOR
        SELECT * FROM home_properties
        WHERE address LIKE '%' || address_in || '%';

    ELSIF site_type_rec.site_type = commercial_type
    THEN
        /* Использование таблицы коммерческих объектов. */
        OPEN site_cur_inout FOR
        SELECT * FROM commercial_properties
        WHERE address LIKE '%' || address_in || '%';
    END IF;
END open_site_list;

```

2. Созданная процедура используется для перебора строений.

В следующем примере я передаю адрес, а затем пытаюсь выполнить выборку из курсора в предположении, что адрес относится к жилому дому. Если адрес в действительности относится к коммерческой недвижимости, PL/SQL инициирует исключение ROWTYPE_MISMATCH из-за несовместимости структур записей. Затем раздел исключений снова осуществляет выборку, на этот раз в запись коммерческого строения, и перебор завершается:

```

/* Файл в Сети: rowtype_mismatch.sql */
DECLARE
    /* Объявление курсорной переменной. */
    building_curvar    sys_refcursor;
    address_string      property_master.address%TYPE;

    /* Определение структур записей для двух разных таблиц. */
    home_rec           home_properties%ROWTYPE;
    commercial_rec      commercial_properties%ROWTYPE;
BEGIN
    /* Получение адреса из cookie или другого источника. */
    address_string := current_address ();

    /* Присваивание запроса курсорной переменной на основании адреса. */
    open_site_list (address_string, building_curvar);

    /* Попытка выборки строки в запись жилого дома. */
    FETCH building_curvar
    INTO home_rec;

    /* Если управление передано в эту точку, адрес относится
       к жилому дому; выводим его. */
    show_home_site (home_rec);
EXCEPTION
    /* Если первая запись не относилась к жилому дому... */
    WHEN ROWTYPE_MISMATCH
    THEN
        /* Выборка той же первой строки в запись коммерческого строения. */
        FETCH building_curvar

```

```
INTO commercial_rec;  
  
/* Вывод информации о коммерческом строении. */  
show_commercial_site (commercial_rec);  
  
END;
```

Правила использования курсорных переменных

В этом разделе более подробно рассматриваются правила и вопросы, связанные с использованием курсорных переменных в программах. Речь пойдет о правилах соответствия типов данных строк, псевдонимах курсорных переменных и области их действия.

Прежде всего помните, что курсорная переменная — это ссылка на объект курсора, представляющий данные, выбранные из базы данных с помощью содержащегося в нем запроса. Это не сам объект курсора. Курсорная переменная может быть связана с определенным запросом при выполнении одного из следующих условий:

- при выполнении команды **OPEN FOR**, назначающий курсорной переменной этот запрос;
- если курсорной переменной присваивается значение другой курсорной переменной, указывающей на этот запрос.

С курсорными переменными может использоваться операция присваивания, и эти переменные могут передаваться в качестве аргументов процедурам и функциям. Чтобы иметь возможность выполнять такие операции, переменные должны удовлетворять правилам соответствия типов строк, проверяемым во время компиляции или во время выполнения, в зависимости от типа курсорных переменных.

Правила соответствия типов строк, проверяемые во время компиляции

В процессе компиляции программы PL/SQL проверяет соблюдение следующих правил:

- Две курсорные переменные (или два параметра) совместимы по присваиванию и передаче аргументов в любом из следующих случаев.
 - обе переменные (или параметра) определены на основе сильнотипизированных форм **REF CURSOR** с одним и тем же именем типа возвращаемой записи;
 - обе переменные (или параметра) определены на основе слаботипизированных форм **REF CURSOR** независимо от типа возвращаемой записи;
 - одна переменная (или параметр) определена на основе сильнотипизированной формы **REF CURSOR**, а другая — на основе слаботипизированной формы.
- Курсорная переменная (или параметр), определенная на основе сильнотипизированной формы **REF CURSOR**, может открыть запрос, который возвращает строку типа, структурно эквивалентного возвращаемому типу из определения типа курсорной переменной.
- Курсорная переменная (или параметр), определенная на основе слаботипизированной формы **REF CURSOR**, может открыть любой запрос. Из этой переменной можно выбирать данные в любой список переменных или в запись любого типа.

Если одна из курсорных переменных определена на основе слаботипизированной формы **REF CURSOR**, компилятор PL/SQL не может проверить совместимость типов возвращаемых строк. Такая проверка будет произведена во время выполнения программы с учетом правил, описанных в следующем разделе.

Правила соответствия типов строк, проверяемые во время выполнения

При выполнении программы PL/SQL проверяет, соблюдаются ли следующие правила:

- Курсорной переменной (или параметру), определенной на основе слаботипизированной формы **REF CURSOR**, можно назначить любой запрос независимо от типа возвращаемых им строк.

- Курсорной переменной (или параметру), определенной на основе сильнотипизированной формы REF CURSOR, можно назначить только запрос, структурно соответствующий типу записи, указанному в секции RETURN определения типа курсорной переменной.
- Две записи (или два списка переменных) считаются структурно совместимыми при выполнении двух условий:
 - количество полей в обеих записях (или количество переменных в обоих списках) одинаково;
 - для каждого поля одной записи (или переменной одного списка) соответствующее поле (или переменная из другого списка) имеет тот же тип данных или может быть неявно преобразовано к тому же типу данных.
- Список переменных и запись в секции INTO команды FETCH должны структурно соответствовать связанному с курсорной переменной запросу. Это правило касается и статических курсоров.

Псевдоним объекта курсора

Если одна курсорная переменная присваивается другой курсорной переменной, то обе они становятся *псевдонимами* одного и того же объекта курсора. В данном случае в результате присваивания в принимающую переменную копируется только ссылка на объект курсора. Результаты любой операции с этим курсором, выполняемые через одну из переменных, сразу же становятся доступными для другой переменной.

Следующий анонимный блок демонстрирует принцип действия псевдонимов курсоров:

```

1  DECLARE
2      TYPE curvar_type IS REF CURSOR;
3      curvar1 curvar_type;
4      curvar2 curvar_type;
5      story fairy_tales%ROWTYPE;
6  BEGIN
7      OPEN curvar1 FOR SELECT * FROM fairy_tales;
8      curvar2 := curvar1;
9      FETCH curvar1 INTO story;
10     FETCH curvar2 INTO story;
11     CLOSE curvar2;
12     FETCH curvar1 INTO story;
13 END;
```

Основные действия, выполняемые этим кодом, описаны в следующей таблице.

| Строки | Описание |
|--------|---|
| 2–5 | Объявление слаботипизированного типа REF CURSOR и курсорных переменных на его основе |
| 7 | Создание объекта курсора и назначение его курсорной переменной curvar1 |
| 8 | Назначение того же объекта курсора второй курсорной переменной curvar2. (Теперь у нас две переменные, которые могут использоваться для работы с одним и тем же результирующим набором строк) |
| 9 | Выборка первой строки с использованием переменной curvar1 |
| 10 | Выборка второй строки с использованием переменной curvar2. (Совершенно не важно, какая из переменных используется для выборки — указатель текущей строки находится в объекте курсора, а не в переменной) |
| 11 | Закрытие объекта курсора с использованием переменной curvar2 |
| 12 | При попытке выборки строки с использованием переменной curvar1 инициируется исключение INVALID_CURSOR. (Когда курсор закрывается через переменную curvar2, он также закрывается и для обращения через переменную curvar1) |

Любое изменения состояния объекта курсора отражается во всех ссылающихся на него переменных.

Область действия объекта курсора

Курсорные переменные обладают такой же областью действия, как и статические курсоры: это блок PL/SQL, в котором объявлена переменная. Однако с областью действия объекта курсора, которому присваивается курсорная переменная, дело обстоит иначе.

Как только команда `OPEN FOR` создаст объект курсора, этот объект остается доступным, пока на него ссылается хотя бы одна активная курсорная переменная. Это означает, что объект курсора можно создать в одной области действия (блок PL/SQL) и присвоить его курсорной переменной. Присваивая эту курсорную переменную другой курсорной переменной с другой областью действия, вы обеспечиваете доступность объекта курсора, даже если исходная курсорная переменная выходит из области действия.

В следующем примере я использую вложенные блоки для демонстрации того, как объект курсора продолжает существовать вне области действия, в которой он был изначально создан:

```
DECLARE
    curvar1 SYS_REFCURSOR;
    do_you_get_it VARCHAR2(100);
BEGIN
    /*
    || Вложенный блок создает объект курсора и присваивает его
    || курсорной переменной curvar1.
    */
    DECLARE
        curvar2 SYS_REFCURSOR;
    BEGIN
        OPEN curvar2 FOR SELECT punch_line FROM joke;
        curvar1 := curvar2;
    END;
    /*
    || Курсорная переменная curvar2 не активна, но "эстафета"
    || была передана переменной curvar1, существующей
    || во внешнем блоке. Соответственно, мы можем выполнить выборку
    || из объекта курсора через другую переменную.
    */
    FETCH curvar1 INTO do_you_get_it;
    CLOSE curvar1;
END;
```

Передача курсорных переменных в аргументах

Курсорную переменную можно передать в качестве аргумента процедуре или функции. В списке параметров такой процедуры (или функции) должен быть задан режим использования и тип `REF CURSOR`.

Идентификация типа `REF CURSOR`

В заголовке программы необходимо идентифицировать тип параметра курсорной переменной. Для этого он должен быть заранее объявлен.

Если вы создаете локальный модуль внутри другой программы (о локальных модулях подробно рассказано в главе 17), тип курсора можно определить в той же программе. Пример:

```
DECLARE
    /* Определение типа REF CURSOR. */
    TYPE curvar_type IS REF CURSOR RETURN company%ROWTYPE;

    /* Тип указывается в списке параметров. */
```

```

PROCEDURE open_query (curvar_out OUT curvar_type)
IS
    local_cur curvar_type;
BEGIN
    OPEN local_cur FOR SELECT * FROM company;
    curvar_out := local_cur;
END;
BEGIN
    ...
END;

```

Если вы создаете отдельную процедуру или функцию, сослаться на существующий тип REF CURSOR можно только одним способом: разместив команду TYPE в пакете. Все переменные, объявленные в спецификации пакета, становятся глобальными в рамках сеанса, и для ссылок на этот тип курсора может использоваться точечный синтаксис. Последовательность действий в этом случае должна быть такой:

- Создаем пакет с объявлением типа REF CURSOR:

```

PACKAGE company
IS
    /* Определение типа REF CURSOR. */
    TYPE curvar_type IS REF CURSOR RETURN company%ROWTYPE;
END package;

```

- В отдельной процедуре ссылаемся на тип REF CURSOR, указывая перед именем типа имя пакета:

```

PROCEDURE open_company (curvar_out OUT company.curvar_type) IS
BEGIN
    ...
END;

```

Назначение режима параметра

Курсорные переменные, как и любые параметры, могут работать в одном из трех режимов:

- IN — разрешается только чтение данных,
- OUT — разрешается только запись данных,
- IN OUT — разрешается как чтение, так и запись данных.

Помните, что значение курсорной переменной представляет собой ссылку на объект курсора, а не состояние этого объекта. Иначе говоря, значение курсорной переменной не изменяется после выборки данных или закрытия курсора.

Значение курсорной переменной (то есть объект курсора, на который указывает переменная) может измениться только в результате выполнения двух операций:

- Присваивание курсорной переменной.
- Выполнение команды OPEN FOR.

Если курсорная переменная уже указывает на объект курсора, OPEN FOR не изменяет ссылку; изменяется только запрос, связанный с объектом.

Операции FETCH и CLOSE изменяют состояние объекта курсора, но не ссылку на объект, которая является значением курсорной переменной.

Пример программы, использующей курсорные переменные в параметрах:

```

PROCEDURE assign_curvar
    (old_curvar_in IN company.curvar_type,
    new_curvar_out OUT company.curvar_type)
IS
BEGIN
    new_curvar_out := old_curvar_in;
END;

```

Процедура копирует старую курсорную переменную в новую переменную. Первый параметр объявлен с режимом **IN**, потому что он используется только в правой части присваивания. Второй параметр должен быть объявлен в режиме **OUT** (или **IN OUT**), потому что его значение изменяется внутри процедуры. Обратите внимание: тип `curvar_type` определяется в пакете `compary`.

Ограничения на использование курсорных переменных

Использование курсорных переменных подчиняется следующим ограничениям (возможно, компания Oracle снимет некоторые из них в будущих версиях):

- Курсорные переменные не могут объявляться в пакетах, потому что они не обладают долгосрочным состоянием.
- Не допускается использование вызовов RPC (Remote Procedure Call) для передачи курсорных переменных между серверами.
- Если курсорная переменная передается в качестве переменной привязки или хост-переменной в PL/SQL, вы не сможете выполнить выборку из нее с сервера, если только она не открывается в вызове с того же сервера.
- Запрос, связанный с курсорной переменной в команде **OPEN FOR**, не может использовать секцию **FOR UPDATE** в Oracle8i и более ранних версиях.
- Курсорные переменные не могут проверяться на равенство, неравенство или неопределенность с использованием операторов сравнения.
- Курсорной переменной нельзя присвоить **NULL**; Oracle выдает сообщение об ошибке *PLS-00382: Expression is of wrong type*.
- Значения курсорных переменных не могут храниться в столбцах базы данных. Тип **REF CURSOR** не может использоваться для определения типа столбцов в командах **CREATE TABLE**.
- Значения курсорных переменных не могут храниться в элементах вложенных таблиц, ассоциативных массивов или **VARRAY**. Тип **REF CURSOR** не может использоваться для задания типа элементов коллекций.

Курсорные выражения

Oracle расширяет язык SQL еще одним мощным элементом: *курсорными выражениями*. Курсорное выражение — это выражение со специальным оператором **CURSOR**, используемое в SQL-запросе и определяющее вложенный курсор. Каждая строка результирующего набора вложенного курсора может содержать диапазон значений, допустимых для SQL-запросов; кроме того, она может включать другие курсоры, определяемые вложенными запросами.



Синтаксис **CURSOR**, введенный в Oracle8i, не мог использоваться в программах PL/SQL. В Oracle9i недостаток был устранен, и теперь курсорные выражения могут применяться в командах SQL программ PL/SQL.

С помощью курсорного выражения можно вернуть большой и сложный набор связанных значений из одной или нескольких таблиц. Такой набор обычно обрабатывается во вложенных циклах: главный цикл выбирает строки основного курсора, а вложенные циклы выбирают строки вложенных курсоров.

Курсорное выражение может быть довольно сложным, что зависит от сложности извлекаемых с его помощью данных. Тем не менее разработчику полезно знать все возможные способы получения данных из баз данных Oracle.

Курсорные выражения могут использоваться в следующих структурных компонентах языка:

- объявления явных курсоров;
- динамические SQL-запросы;
- объявления и переменные типа REF CURSOR.

В неявных курсорах курсорные выражения никогда не используются.

Синтаксис курсорного выражения очень прост:

`CURSOR (вложенный_запрос)`

Oracle неявно открывает вложенный курсор при выборке строки, содержащей его выражение из родительского или внешнего курсора. Он закрывается, когда:

- вы явно закрываете курсор;
- родительский курсор повторно выполняется, закрывается или отменяется;
- при выборке из родительского курсора инициируется исключение. В этом случае вложенный курсор закрывается вместе с родительским.

Использование курсорных выражений

Существует два разных, но очень полезных способа использования курсорных выражений:

- для выборки вложенного запроса как столбца внешнего запроса;
- для преобразования запроса в результирующий набор, который может передаться в аргументе функции.

Выборка вложенного запроса как столбца

Следующая процедура демонстрирует первый способ использования вложенного курсорного выражения. Запрос верхнего уровня выбирает два элемента данных: название города и отделение компании в этом городе, информация о котором содержится во вложенном курсоре. Этот вложенный курсор, в свою очередь, извлекает с помощью курсорного выражения еще один вложенный курсор — на этот раз с фамилиями всех сотрудников каждого отделения.

Те же данные можно было бы извлечь из базы данных с помощью нескольких явных курсоров, открываемых и обрабатываемых во вложенных циклах. Однако курсорное выражение позволяет применить другой подход, более лаконичный и эффективный (поскольку вся обработка происходит в исполнителе команд SQL, что сокращает количество переключений контекста):

```
PROCEDURE emp_report (p_locid NUMBER)
IS
    TYPE refcursor IS REF CURSOR;

    -- Запрос возвращает 2 столбца, второй из которых является курсором,
    -- а значит, позволяет перебрать набор связанных строк.
    CURSOR all_in_one_cur is
        SELECT l.city,
               CURSOR (SELECT d.department_name,
                             CURSOR(SELECT e.last_name
                                     FROM employees e
                                     WHERE e.department_id =
                                           d.department_id)
                             AS ename
               FROM departments d
               WHERE l.location_id = d.location_id) AS dname
```

продолжение ➤

```

        FROM locations l
        WHERE l.location_id = p_locid;
    departments_cur    refcursor;
    employees_cur      refcursor;

    v_city      locations.city%TYPE;
    v_dname     departments.department_name%TYPE;
    v_ename     employees.last_name%TYPE;
BEGIN
    OPEN all_in_one_cur;

    LOOP
        FETCH all_in_one_cur INTO v_city, departments_cur;
        EXIT WHEN all_in_one_cur%NOTFOUND;

        -- Перебираем список отделений, причем для этого НЕ НУЖНО
        -- явно открывать курсор. Oracle делает это автоматически.
        LOOP
            FETCH departments_cur INTO v_dname, employees_cur;
            EXIT WHEN departments_cur%NOTFOUND;

            -- Теперь можно перебрать всех сотрудников текущего
            -- отделения. И снова явно открывать курсор не нужно.
            LOOP
                FETCH employees_cur INTO v_ename;
                EXIT WHEN employees_cur%NOTFOUND;
                DBMS_OUTPUT.put_line (
                    v_city
                    ||
                    || v_dname
                    ||
                    || v_ename
                );
            END LOOP;
        END LOOP;
    END LOOP;
    CLOSE all_in_one_cur;
END;
```

Реализация потоковых функций в выражениях CURSOR

Потоковые функции, также называемые *преобразующими* функциями, позволяют преобразовать данные из одного состояния в другое без использования локальных структур данных для промежуточного хранения результатов. Предположим, нужно взять данные из таблицы `StockTable` и переместить их в таблицу `TickerTable`, преобразуя одну строку `StockTable` в две строки `TickerTable`. С курсорными выражениями и табличными функциями решение может быть реализовано следующим образом:

```

INSERT INTO TickerTable
SELECT *
FROM TABLE (StockPivot (CURSOR (SELECT * FROM StockTable)));
```

Функция `StockPivot` содержит всю сложную логику, необходимую для выполнения преобразования. Этот прием более подробно рассматривается в главе 17.

Ограничения, связанные с курсорными выражениями

Для применения курсорных выражений установлены некоторые ограничения:

- Курсорное выражение не может использоваться в неявном курсоре по причине отсутствия механизма выборки из вложенного курсора в структуру данных PL/SQL.

- Курсорные выражения могут задаваться только в «самом внешнем» списке выборки запроса.
- Курсорные выражения нельзя использовать в команде `SELECT`, вложенной в другой запрос (исключение составляет вложенный запрос курсорного выражения).
- Курсорные выражения не могут использоваться при объявлении представления.
- Если курсорное выражение задано в динамической команде `SQL`, к нему нельзя применять операции `BIND` и `EXECUTE` (см. главу 16).

16

Динамический SQL и динамический PL/SQL

Термином «*динамический SQL*» обозначаются команды SQL, которые конструируются и вызываются непосредственно во время выполнения программы. *Статическими* называются жестко закодированные команды SQL, которые не изменяются с момента компиляции программы. «*Динамическим PL/SQL*» называют целые блоки кода PL/SQL, которые строятся динамически, а затем компилируются и выполняются.

Пожалуй, написание динамических команд SQL и динамических программ PL/SQL было самым интересным делом из всего, что я когда-либо делал на языке PL/SQL. Конструирование и динамическое выполнение обеспечивает невероятную гибкость. У разработчика появляется возможность создавать обобщенный код с широким спектром применения. Несмотря на это, динамический SQL следует применять лишь там, где это необходимо; решения со статическим SQL всегда являются предпочтительными, потому что динамические решения более сложны, создают больше проблем с отладкой и тестированием, обычно медленнее работают и усложняют сопровождение.

Что же можно делать с динамическими конструкциями SQL и PL/SQL?¹ Лишь несколько идей:

- **Выполнение команд DDL.** Со статическим кодом SQL в PL/SQL могут выполняться только запросы и команды DML. А если вы захотите создать таблицу или удалить индекс? Используйте динамический SQL!
- **Поддержка специализированных запросов и требований к обновлению веб-приложений.** К интернет-приложениям часто предъявляется одно стандартное требование: пользователь должен иметь возможность выбрать столбцы, которые он желает видеть, и изменить порядок просмотра данных (конечно, пользователь может и не понимать, что именно при этом происходит).
- **Оперативное изменение бизнес-правил и формул.** Вместо того чтобы жестко фиксировать бизнес-правила в коде, можно разместить соответствующую логику в таблицах. Во время выполнения программа генерирует и выполняет код PL/SQL, необходимый для применения правил.

Начиная с Oracle7, поддержка динамического SQL осуществлялась в виде встроенного пакета DBMS_SQL. В Oracle8i для этого появилась еще одна возможность — встроенный динамический SQL (Native Dynamic SQL, NDS). NDS интегрируется в язык PL/SQL; пользоваться им намного удобнее, чем DBMS_SQL. Впрочем, для некоторых ситуаций

¹ В оставшейся части этой главы все упоминания «динамического SQL» также подразумевают динамические блоки PL/SQL (если обратное явно не указано в тексте).

лучше подходит DBMS_SQL; в конце главы мы сравним оба средства и приведем некоторые рекомендации по их использованию. На практике в подавляющем большинстве случаев NDS является более предпочтительным решением.

Команды NDS

Главным достоинством NDS является его простота. В отличие от пакета DBMS_SQL, для работы с которым требуется знание десятка программ и множества правил их использования, NDS представлен в PL/SQL единственной новой командой EXECUTE IMMEDIATE, которая немедленно выполняет заданную команду SQL, а также расширением существующей команды OPEN FOR, позволяющей выполнять многострочные динамические запросы.



Команды EXECUTE IMMEDIATE и OPEN FOR не будут напрямую доступны в Oracle Forms Builder и Oracle Reports Builder до тех пор, пока версия PL/SQL этих инструментов не будет обновлена до Oracle8i и выше. Для более ранних версий придется создавать хранимые программы, скрывающие вызовы этих конструкций; эти хранимые программы могут выполняться в клиентском коде PL/SQL.

Команда EXECUTE IMMEDIATE

Команда EXECUTE IMMEDIATE используется для немедленного выполнения заданной команды SQL. Она имеет следующий синтаксис:

```
EXECUTE IMMEDIATE строка_SQL
  [ [ BULK COLLECT ] INTO { переменная [, переменная] ... | запись }
  [ USING [ IN | OUT | IN OUT ] аргумент
    [, [ IN | OUT | IN OUT ] аргумент] ... ];
```

Здесь *строка_SQL* — строковое выражение, содержащее команду SQL или блок PL/SQL; *переменная* — переменная, которой присваивается содержимое поля, возвращаемого запросом; *запись* — запись, основанная на пользовательском типе или типе %ROWTYPE, принимающая всю возвращаемую запросом строку; *аргумент* — либо выражение, значение которого передается команде SQL или блоку PL/SQL, либо идентификатор, являющийся входной и/или выходной переменной для функции или процедуры, вызываемой из блока PL/SQL. Секция INTO используется для однострочных запросов. Для каждого значения столбца, возвращаемого запросом, необходимо указать переменную или поле записи совместимого типа. Если INTO предшествует конструкции BULK COLLECT, появляется возможность выборки множественных строк в одну или несколько коллекций. Секция USING предназначена для передачи аргументов строке SQL. Она используется с динамическим SQL и PL/SQL, что и позволяет задать режим параметра. Этот режим актуален только для PL/SQL и секции RETURNING. По умолчанию для параметров используется режим IN (для команд SQL допустима только эта разновидность аргументов).

Команда EXECUTE IMMEDIATE может использоваться для выполнения любой команды SQL или блока PL/SQL. Строка может содержать формальные параметры, но они не могут связываться с именами объектов схемы (например, именами таблиц или столбцов).



При выполнении команды DDL в программе также происходит закрепление операции. Если вы не хотите, чтобы закрепление, обусловленное DDL, отражалось на текущих изменениях в других частях приложения, поместите динамическую команду DDL в процедуру автономной транзакции. Пример такого рода приведен в файле auton_ddl.sql на сайте книги.

При выполнении команды исполняющее ядро заменяет в SQL-строке формальные параметры (идентификаторы, начинающиеся с двоеточия — например, :salary_value)

фактическими значениями параметров подстановки в секции USING. Не допускается передача литерала NULL — вместо него следует указывать выражение соответствующего типа, результат вычисления которого может быть равен NULL.

NDS поддерживает все типы данных SQL. Переменные и параметры команды могут быть коллекциями, большими объектами (LOB), экземплярами объектных типов, документами XML и т. д. Однако NDS не поддерживает типы данных, специфические для PL/SQL, такие как BOOLEAN, ассоциативные массивы и пользовательские типы записей. С другой стороны, секция INTO может содержать запись PL/SQL, количество и типы полей которой соответствуют значениям, выбранным динамическим запросом.

Рассмотрим несколько примеров.

- Создание индекса:

```
BEGIN
  EXECUTE IMMEDIATE 'CREATE INDEX emp_u_1 ON employees (last_name)';
END;
```

Проще не бывает, верно?

- Создание хранимой процедуры, выполняющей любую команду DDL:

```
PROCEDURE exec_DDL (ddl_string IN VARCHAR2)
IS
BEGIN
  EXECUTE IMMEDIATE ddl_string;
END;
```

При наличии процедуры exec_ddl тот же индекс может быть создан следующим образом:

```
BEGIN
  exec_DDL ('CREATE INDEX emp_u_1 ON employees (last_name)');
END;
Получение количества строк в произвольной таблице для заданного предложения WHERE:
/* Файл в Сети: tabcount_nds.sf */
FUNCTION tabcount (table_in IN VARCHAR2)
  RETURN PLS_INTEGER
IS
  l_query VARCHAR2 (32767) := 'SELECT COUNT(*) FROM ' || table_in;
  l_return PLS_INTEGER;
BEGIN
  EXECUTE IMMEDIATE l_query INTO l_return;
  RETURN l_return;
END;
```

Таким образом, нам больше не понадобится писать команду SELECT COUNT(*) ни в SQL*Plus, ни в программах PL/SQL. Она заменяется следующим блоком кода:

```
BEGIN
  IF tabCount ('employees') > 100
  THEN
    DBMS_OUTPUT.PUT_LINE ('We are growing fast!');
  END IF;
END;
```

Изменение числового значения в любом столбце таблицы employees:

```
/* Файл в Сети: updNval.sf */
FUNCTION updNval (
  col IN VARCHAR2,
  val IN NUMBER,
  start_in IN DATE,
  end_in IN DATE)
  RETURN PLS_INTEGER
IS
BEGIN
  EXECUTE IMMEDIATE
    'UPDATE employees SET ' || col || ' = :the_value
```

```

        WHERE hire_date BETWEEN :lo AND :hi'
        USING val, start_in, end_in;
    RETURN SQL%ROWCOUNT;
END;
```

Безусловно, для такой гибкости объем кода получился совсем небольшим! В этом примере показано, как используется подстановка: после разбора команды UPDATE ядро PL/SQL заменяет в ней формальные параметры (:the_value, :lo и :hi) значениями переменных. Также обратите внимание, что в этом случае атрибут курсора SQL%ROWCOUNT используется точно так же, как при выполнении статических команд DML.

Выполнение разных блоков кода в одно время в разные дни. Имя каждой программы строится по схеме *ДЕНЬ_set_schedule*. Все процедуры получают четыре аргумента: при вызове передается код работника employee_id и час первой встречи, а процедура возвращает имя работника и количество встреч в заданный день. Задача решается с использованием динамического PL/SQL:

```

/* Файл в Сети: run9am.sp */
PROCEDURE run_9am_procedure (
    id_in IN employee.employee_id%TYPE,
    hour_in IN INTEGER)
IS
    v_apptCount INTEGER;
    v_name VARCHAR2(100);
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN ' || TO_CHAR (SYSDATE, 'DAY') ||
        '_set_schedule (:id, :hour, :name, :appts); END;'
    USING IN
        id_in, IN hour_in, OUT v_name, OUT v_apptCount;
    DBMS_OUTPUT.PUT_LINE (
        'Employee ' || v_name || ' has ' || v_apptCount ||
        ' appointments on ' || TO_CHAR (SYSDATE));
END;
```

- Привязка значения BOOLEAN, специфического для PL/SQL, командой EXECUTE IMMEDIATE (новая возможность 12c):

```

/* Файл в Сети: 12c_bind_boolean.sql */
CREATE OR REPLACE PACKAGE restaurant_pkg
AS
    TYPE item_list_t
        IS TABLE OF VARCHAR2 (30);

    PROCEDURE eat_that (
        items_in          IN item_list_t,
        make_it_spicy_in_in IN BOOLEAN);
END;
/

CREATE OR REPLACE PACKAGE BODY restaurant_pkg
AS
    PROCEDURE eat_that (
        items_in          IN item_list_t,
        make_it_spicy_in_in IN BOOLEAN)
    IS
    BEGIN
        FOR indx IN 1 .. items_in.COUNT
        LOOP
            DBMS_OUTPUT.put_line (
                CASE
                    WHEN make_it_spicy_in_in
                    THEN
                        'Spicy '
                END
            );
        END LOOP;
    END;
```

```

        || items_in (indx));
    END LOOP;
END;
END;
/

DECLARE
    things    restaurant_pkg.item_list_t
        := restaurant_pkg.item_list_t (
            'steak',
            'quiche',
            'eggplant');
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN restaurant_pkg.eat_that(:1, :s); END;'
        USING things, TRUE;
END;
/

```

Как видите, команда `EXECUTE IMMEDIATE` позволяет исключительно легко выполнять динамические команды SQL и блоки PL/SQL с удобным синтаксисом.

Команда OPEN FOR

Команда `OPEN FOR` изначально *не была* включена в PL/SQL для NDS; она появилась в Oracle7 и предназначалась для работы с курсорными переменными. Затем ее синтаксис был расширен для реализации многострочных динамических запросов. При использовании пакета `DBMS_SQL` реализация многострочных запросов получается очень сложной: приходится производить разбор и подстановку, отдельно определять каждый столбец, выполнять команду, выбирать сначала строки, а затем — последовательно значения каждого столбца. Код получается весьма громоздким.

Для динамического SQL разработчики Oracle сохранили существующий синтаксис `OPEN`, но расширили его вполне естественным образом:

```

OPEN {курсорная_переменная} :хост_переменная} FOR строка_SQL
    [USING аргумент [, аргумент]...];

```

Здесь *курсорная_переменная* — слаботипизированная курсорная переменная; *хост_переменная* — курсорная переменная, объявленная в хост-среде PL/SQL, например в программе OCI (Oracle Call Interface); *строка_SQL* — команда `SELECT`, подлежащая динамическому выполнению.

Курсорные переменные рассматриваются в главе 15. В этой главе мы подробно расскажем об их использовании с NDS.

В следующем примере объявляется тип `REF CURSOR` и основанная на нем переменная-курсор, а затем с помощью команды `OPEN FOR` открывается динамический запрос:

```

PROCEDURE show_parts_inventory (
    parts_table IN VARCHAR2,
    where_in IN VARCHAR2)
IS
    TYPE query_curtype IS REF CURSOR;
    dyncur query_curtype;
BEGIN
    OPEN dyncur FOR
        'SELECT * FROM ' || parts_table
        ' WHERE ' || where_in;
    ...

```

После того как запрос будет открыт командой `OPEN FOR`, синтаксис выборки записи, закрытия курсорной переменной и проверки атрибутов курсора ничем не отличается от синтаксиса статических курсорных переменных и явных курсоров.

Давайте поближе познакомимся с командой OPEN FOR. При выполнении OPEN FOR ядро PL/SQL:

- 1) связывает курсорную переменную с командой SQL, заданной в строке запроса;
- 2) вычисляет значения параметров и заменяет ими формальные параметры в строке запроса;
- 3) выполняет запрос;
- 4) идентифицирует результирующий набор;
- 5) устанавливает курсор на первую строку результирующего набора;
- 6) обнуляет счетчик обработанных строк, возвращаемый атрибутом %ROWCOUNT.

Обратите внимание: параметры подстановки, заданные в секции USING, вычисляются только при открытии курсора. Это означает, что для передачи тому же динамическому запросу другого набора параметров нужно выполнить новую команду OPEN FOR.

Для выполнения многострочного запроса (то есть запроса, возвращающего набор строк) необходимо:

- 1) объявить тип REF CURSOR (или использовать встроенный тип SYS_REFCURSOR);
- 2) объявить на его основе курсорную переменную;
- 3) открыть курсорную переменную командой OPEN FOR;
- 4) с помощью команды FETCH по одной извлечь записи результирующего набора;
- 5) при необходимости проверить значения атрибутов (%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN);
- 6) закрыть курсорную переменную обычной командой CLOSE. Как правило, после завершения работы с курсорной переменной следует явно закрыть ее.

Следующая простая программа выводит значения поля заданной таблицы в строках, отбираемых с помощью секции WHERE (столбец может содержать числа, даты или строки):

```
/* Файл в Сети: showcol.sp */
PROCEDURE showcol (
    tab IN VARCHAR2,
    col IN VARCHAR2,
    whr IN VARCHAR2 := NULL)
IS
    cv SYS_REFCURSOR;
    val VARCHAR2(32767);
BEGIN
    OPEN cv FOR
        'SELECT ' || col ||
        ' FROM ' || tab ||
        ' WHERE ' || NVL (whr, '1 = 1');

    LOOP
        /* Выбираем следующую строку; если строк больше нет, цикл завершается. */
        FETCH cv INTO val;
        EXIT WHEN cv%NOTFOUND;

        /* Перед первой строкой выводится заголовок. */
        IF cv%ROWCOUNT = 1
        THEN
            DBMS_OUTPUT.PUT_LINE (RPAD ('-', 60, '-'));
            DBMS_OUTPUT.PUT_LINE (
                'Contents of ' || UPPER (tab) || ' ' || ' ' || UPPER (col));
            DBMS_OUTPUT.PUT_LINE (RPAD ('-', 60, '-'));
        END IF;

        DBMS_OUTPUT.PUT_LINE (val);
    END LOOP;
```

```

/* Закрыть курсор! Это очень важно... */
CLOSE cv;
END;

```

Примерный результат выполнения этой процедуры выглядит так:

```
SQL> EXEC showcol ('emp', 'ename', 'deptno=10')
```

```
-----
Contents of EMP.ENAME
-----
```

```
CLARK
KING
MILLER
```

Столбцы даже можно комбинировать:

```

BEGIN
  showcol (
    'emp',
    'ename || ''-$'' || sal',
    'comm IS NOT NULL');END;/

```

```
-----
Contents of EMP.ENAME || ''-$'' || SAL
-----
```

```
ALLEN-$1600
WARD-$1250
MARTIN-$1250
TURNER-$1500
```

Выборка в переменные или записи

Команда `FETCH` в процедуре `showcol` из предыдущего раздела осуществляет выборку в отдельную переменную. Также возможна выборка в серию переменных:

```

PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
  cv SYS_REFCURSOR;
  mega_bucks company.ceo_compensation%TYPE;
  achieved_by company.cost_cutting%TYPE;
BEGIN
  OPEN cv FOR
    'SELECT ceo_compensation, cost_cutting
     FROM ' || company_table_name (company_id_in);

  LOOP
    FETCH cv INTO mega_bucks, achieved_by;
    ...
  END LOOP;

  CLOSE cv;
END;

```

Работа с длинным списком переменных в списке `FETCH` может быть громоздкой и недостаточно гибкой; вы должны объявить переменные, поддерживать синхронизацию этого набора значений в команде `FETCH` и т. д. Чтобы упростить жизнь разработчика, NDS позволяет осуществить выборку в запись, как показано в следующем примере:

```

PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
  cv SYS_REFCURSOR;
  ceo_info company%ROWTYPE;
BEGIN
  OPEN cv FOR
    'SELECT * FROM ' || company_table_name (company_id_in);

  LOOP

```

```

        FETCH cv INTO ceo_info;
        ...
    END LOOP;

    CLOSE cv;
END;
```

Конечно, во многих ситуациях выполнение команды **SELECT *** нежелательно; если ваша таблица содержит сотни столбцов, из которых вам нужны два-три, эта команда крайне неэффективна. Лучше создать тип записи, соответствующий разным требованиям. Эти структуры лучше всего разместить в спецификации пакета, чтобы их можно было использовать во всем приложении. Вот один из таких пакетов:

```

PACKAGE company_pkg
IS
    TYPE ceo_info_rt IS RECORD (
        mega_bucks company.ceo_compensation%TYPE,
        achieved_by company.cost_cutting%TYPE);
END company_pkg;
```

С таким пакетом приведенный выше код можно переписать следующим образом:

```

PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
    cv SYS_REFCURSOR;
    rec company_pkg.ceo_info_rt;
BEGIN
    OPEN cv FOR
        'SELECT ceo_compensation, cost_cutting FROM ' ||
        company_table_name (company_id_in);

    LOOP
        FETCH cv INTO rec;
        ...
    END LOOP;

    CLOSE cv;
END;
```

Секция USING в OPEN FOR

Как и в случае с командой **EXECUTE IMMEDIATE**, при открытии курсора можно передать аргументы. Для запроса можно передать только аргументы **IN**. Аргументы также повышают эффективность **SQL**, упрощая написание и сопровождение кода. Кроме того, они могут радикально сократить количество разобранных команд, хранящихся в общей памяти **SGA**, а это повышает вероятность того, что уже разобранная команда будет находиться в **SGA** в следующий раз, когда она вам потребуется. (За дополнительной информацией обращайтесь к разделу «Передача параметров» этой главы.)

Вернемся к процедуре **showcol**. Эта процедура получает полностью обобщенную секцию **WHERE**. Допустим, действуют более специализированные требования: я хочу вывести (или иным образом обработать) всю информацию столбцов для строк, содержащих столбец даты со значением из некоторого диапазона. Другими словами, требуется обеспечить поддержку запроса:

```

SELECT last_name
FROM employees
WHERE hire_date BETWEEN x AND y;
```

а также запроса:

```

SELECT flavor
FROM favorites
WHERE preference_period BETWEEN x AND y;
```

Также нужно проследить за тем, чтобы компонент времени столбца даты не учитывался в условии WHERE.

Заголовок процедуры выглядит так:

```
/* Файл в Сети: showdtcol.sp */
PROCEDURE showcol (
    tab IN VARCHAR2,
    col IN VARCHAR2,
    dtcol IN VARCHAR2,
    dt1 IN DATE,
    dt2 IN DATE := NULL)
```

Теперь команда OPEN FOR содержит два формальных параметра и соответствующую секцию USING:

```
OPEN cv FOR
    'SELECT ' || col ||
    ' FROM ' || tab ||
    ' WHERE ' || dtcol ||
    ' BETWEEN TRUNC (:startdt)
      AND TRUNC (:enddt)'
    USING dt1, NVL (dt2, dt1+1);
```

Команда построена таким образом, что при отсутствии конечной даты секция WHERE возвращает строки, у которых значение в столбце даты совпадает с заданным значением dt1. Остальной код процедуры showcol остается неизменным, не считая косметических изменений в выводе заголовка.

Следующий вызов новой версии showcol запрашивает имена всех работников, принятых на работу в 1982 году:

```
BEGIN
    showcol ('emp', 'ename', 'hiredate',
        DATE '1982-01-01', DATE '1982-12-31');
END;
```

Результат:

```
-----
Contents of EMP.ENAME for HIREDATE between 01-JAN-82 and 31-DEC-82
-----
MILLER
```

О четырех категориях динамического SQL

Итак, мы рассмотрели две основные команды, используемые для реализации динамического SQL в PL/SQL. Теперь пришло время сделать шаг назад и рассмотреть четыре разновидности (*категории*) динамического SQL, а также команды NDS, необходимые для реализации этих категорий. Категории и соответствующие команды NDS перечислены в табл. 16.1.

Таблица 16.1. Четыре категории динамического SQL

| Тип | Описание | Команды NDS |
|---------------------------|--|---|
| Категория 1 | Без запросов; только команды DDL и команды UPDATE, INSERT, MERGE и DELETE без параметров | EXECUTE IMMEDIATE без секций USING и INTO |
| Категория 2 | Без запросов; только команды DDL и команды UPDATE, INSERT, MERGE и DELETE с фиксированным количеством параметров | EXECUTE IMMEDIATE с секцией USING |
| Категория 3 (одна строка) | Запросы (SELECT) с фиксированным количеством столбцов и параметров, с выборкой одной строки данных | EXECUTE IMMEDIATE с секциями USING и INTO |

| Тип | Описание | Команды NDS |
|----------------------------------|--|--|
| Категория 3 (несколько строк) | Запросы (SELECT) с фиксированным количеством столбцов и параметров, с выборкой одной или нескольких строк данных | EXECUTE IMMEDIATE с секциями USING и BULK COLLECT INTO или OPEN FOR с динамической строкой |
| Категория 4 | Команда, в которой количество выбранных столбцов (для запроса) или количество параметров неизвестно до стадии выполнения | Для категории 4 необходим пакет DBMS_SQL |

Категория 1

Следующая команда DDL является примером динамического SQL категории 1:

```
EXECUTE IMMEDIATE 'CREATE INDEX emp_ind_1 on employees (salary, hire_date)';
```

Команда UPDATE также относится к динамическому SQL категории 1, потому что единственным изменяемым аспектом является имя таблицы — параметры отсутствуют:

```
EXECUTE IMMEDIATE
  'UPDATE ' || l_table || ' SET salary = 10000 WHERE employee_id = 1506'
```

Категория 2

Если заменить оба жестко фиксированных значения в предыдущей команде DML формальными параметрами (двоеточие, за которым следует идентификатор), появляется динамический SQL категории 2:

```
EXECUTE IMMEDIATE
  'UPDATE ' || l_table || '
    SET salary = :salary WHERE employee_id = :employee_id'
  USING 10000, 1506;
```

Секция USING содержит значения, которые будут подставлены в строку SQL после разбора и перед выполнением.

Категория 3

Команда динамического SQL категории 3 представляет собой запрос с фиксированным количеством параметров (или вообще без них). Вероятно, чаще всего вы будете создавать команды динамического SQL именно этого типа. Пример:

```
EXECUTE IMMEDIATE
  'SELECT last_name, salary FROM employees
    WHERE department_id = :dept_id'
  INTO l_last_name, l_salary
  USING 10;
```

Здесь я запрашиваю всего два столбца из таблицы `employees` и сохраняю их значения в двух локальных переменных из секции INTO. Также используется один параметр. Так как значения этих компонентов являются статическими на стадии компиляции, я использую динамический SQL категории 3.

Категория 4

Наконец, рассмотрим самый сложный случай: динамический SQL категории 4. Возьмем предельно обобщенный запрос:

```
OPEN l_cursor FOR
  'SELECT ' || l_column_list ||
  'FROM employees';
```

На момент компиляции кода я понятия не имею, сколько столбцов будет запрашиваться из таблицы `employees`. Возникает проблема: как написать команду FETCH INTO,

которая будет обеспечивать подобную изменчивость? Есть два варианта: либо вернуться к DBMS_SQL для написания относительно тривиального (хотя и объемистого) кода, либо переключиться на исполнение динамических блоков PL/SQL.

К счастью, ситуации, требующие применения категории 4, встречаются редко. Но если вы столкнетесь с такой необходимостью, прочитайте раздел «Поддержка требований категории 4» этой главы.

Передача параметров

Мы рассмотрели несколько примеров использования параметров с NDS. Давайте познакомимся с различными правилами и специальными ситуациями, которые могут вам встретиться при передаче параметров.

SQL-запросу могут передаваться только выражения (литералы, переменные, сложные выражения), заменяющие формальные параметры в строке значениями данных. Не допускается передача имен элементов схемы (таблиц, столбцов и т. д.) или целых фрагментов кода SQL (например, условий WHERE), которые должны строиться посредством конкатенации.

Допустим, вы хотите создать процедуру для очистки заданного представления или таблицы. Первая версия может выглядеть примерно так:

```
PROCEDURE truncobj (
    nm IN VARCHAR2,
    tp IN VARCHAR2 := 'TABLE',
    sch IN VARCHAR2 := NULL)
IS
BEGIN
    EXECUTE IMMEDIATE
        'TRUNCATE :trunc_type :obj_name'
        USING tp, NVL (sch, USER) || '.' || nm;
END;
```

На первый взгляд все выглядит вполне разумно, но при попытке выполнения этой процедуры вы получите сообщение об ошибке:

ORA-03290: Invalid truncate command - missing CLUSTER or TABLE keyword

При упрощении процедуры до следующего вида:

```
EXECUTE IMMEDIATE 'TRUNCATE TABLE :obj_name' USING nm;
```

сообщение об ошибке изменится:

ORA-00903: invalid table name

Почему же в NDS (как, впрочем, и в пакете DBMS_SQL) имеется такое ограничение? При передаче строки команде EXECUTE IMMEDIATE исполняющее ядро должно прежде всего выполнить синтаксический анализ команды, чтобы убедиться в правильности ее определения. PL/SQL может определить, что следующая команда определена правильно, даже не зная значения параметра :xyz:

```
'UPDATE emp SET sal = :xyz'
```

Но корректность следующей команды PL/SQL проверить не сможет:

```
'UPDATE emp SET :col_name = :xyz'
```

По этой причине в данной ситуации необходимо использовать конкатенацию:

```
PROCEDURE truncobj (
    nm IN VARCHAR2,
    tp IN VARCHAR2 := 'TABLE',
    sch IN VARCHAR2 := NULL)
IS
```

```
BEGIN
  EXECUTE IMMEDIATE
    'TRUNCATE ' || tp || ' ' || NVL (sch, USER) || '.' || nm;
END;
```

Режимы передачи параметров

При передаче значений параметров команды SQL можно использовать один из трех режимов: IN (только чтение, действует по умолчанию), OUT (только запись) или IN OUT (чтение и запись). При выполнении динамического запроса все параметры команды SQL, за исключением параметра в секции RETURNING, должны передаваться в режиме IN:

```
PROCEDURE wrong_incentive (
  company_in IN INTEGER,
  new_layoffs IN NUMBER
)
IS
  sql_string VARCHAR2(32767);
  sal_after_layoffs NUMBER;
BEGIN
  sql_string :=
    'UPDATE ceo_compensation
     SET salary = salary + 10 * :layoffs
     WHERE company_id = :company
     RETURNING salary INTO :newsal';
  EXECUTE IMMEDIATE sql_string
    USING new_layoffs, company_in, OUT sal_after_layoffs;

  DBMS_OUTPUT.PUT_LINE (
    'CEO compensation after latest round of layoffs $' || sal_after_layoffs);
END;
```

Параметры подстановки команды SQL, передаваемые в режимах OUT и IN OUT, используются прежде всего при выполнении динамического PL/SQL. В этом случае режимы передачи параметров соответствуют аналогичным режимам обыкновенных программ PL/SQL, а также использованию переменных в динамических блоках PL/SQL.

Несколько общих рекомендаций, касающихся использования секции USING при выполнении динамического PL/SQL:

- В качестве параметра подстановки, передаваемого в режиме IN, может быть задан любой элемент соответствующего типа: литеральное значение, именованная константа, переменная или сложное выражение. Такой элемент сначала вычисляется, а затем передается в динамический блок PL/SQL.
- Для значения параметра динамической команды в режиме OUT или IN OUT следует объявить переменную.
- Значения можно подставлять только вместо тех параметров динамического блока PL/SQL, тип которых поддерживается SQL. Например, если параметр процедуры имеет тип BOOLEAN, его значение нельзя задать или считать с помощью секции USING.



Это ограничение частично снято в версии 12.1 и выше. Теперь разрешается подстановка многих типов PL/SQL, включая типы записей и коллекций, но подстановка BOOLEAN по-прежнему запрещена.

Давайте рассмотрим механизм передачи параметров на примерах. Вот заголовок процедуры с параметрами IN, OUT и IN OUT:

```
PROCEDURE analyze_new_technology (
    tech_name IN VARCHAR2,
    analysis_year IN INTEGER,
    number_of_adherents IN OUT NUMBER,
    projected_revenue OUT NUMBER
)
```

Пример блока с динамическим вызовом этой процедуры:

```
DECLARE
    devoted_followers NUMBER;
    est_revenue NUMBER;
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN
            analyze_new_technology (:p1, :p2, :p3, :p4); END;'
    USING 'Java', 2002, IN OUT devoted_followers, OUT est_revenue;
END;
```

Поскольку процедура имеет четыре параметра, в секции `USING` также должно быть указано четыре элемента. Для первых двух параметров, передаваемых в режиме `IN`, следует задать литеральные значения или выражения, а следующие два элемента должны быть именами переменных, так как для них заданы режимы `OUT` и `IN OUT`.

Но что, если два и более формальных параметра имеют одинаковые имена?

Дублирование формальных параметров

При выполнении динамической команды SQL связь между формальными и фактическими параметрами устанавливается в соответствии с их *позициями*. Однако интерпретация одноименных параметров зависит от того, какой код используется — SQL или PL/SQL.

- При выполнении динамической команды SQL (DML- или DDL-строки, *не* оканчивающейся точкой с запятой) параметр подстановки нужно задать для каждого формального параметра, даже если их имена повторяются.
- При выполнении динамического блока PL/SQL (строки, оканчивающейся точкой с запятой) нужно указать параметр подстановки для каждого *уникального* формального параметра.

Далее приведен пример динамической команды SQL с повторяющимися формальными параметрами. Особое внимание обратите на повторяющийся параметр подстановки `val_in` в секции `USING`:

```
PROCEDURE updnunval (
    col_in    IN    VARCHAR2,
    start_in  IN    DATE, end_in  IN    DATE,
    val_in    IN    NUMBER)
IS
    dml_str VARCHAR2(32767) :=
        'UPDATE emp SET ' || col_in || ' = :val
        WHERE hiredate BETWEEN :lodate AND :hidate
        AND :val IS NOT NULL';
BEGIN
    EXECUTE IMMEDIATE dml_str
    USING val_in, start_in, end_in, val_in;
END;
```

А вот динамический блок PL/SQL с повторяющимися формальными параметрами — для него в секции `USING` параметр `val_in` задан только один раз:

```
PROCEDURE updnunval (
    col_in    IN    VARCHAR2,
    start_in  IN    DATE, end_in  IN    DATE,
    val_in    IN    NUMBER)
IS
```

```

dml_str VARCHAR2(32767) :=
'BEGIN
    UPDATE emp SET ' || col_in || ' = :val
    WHERE hiredate BETWEEN :lodate AND :hidate
    AND :val IS NOT NULL;
END;';
BEGIN
    EXECUTE IMMEDIATE dml_str
    USING val_in, start_in, end_in;
END;

```

Передача значений NULL

При передаче NULL в качестве параметра подстановки — например, как в команде:

```

EXECUTE IMMEDIATE
'UPDATE employee SET salary = :newsal
  WHERE hire_date IS NULL'
USING NULL;

```

вы получите сообщение об ошибке. Дело в том, что NULL не имеет типа данных и поэтому не может являться значением одного из типов данных SQL.

Что же делать, если вам потребуется передать в динамический код значение NULL? Это можно сделать двумя способами.

Во-первых, значение можно скрыть в переменной, для чего проще всего использовать неинициализированную переменную подходящего типа:

```

DECLARE
    /* Исходным значением переменной по умолчанию является NULL */
    no_salary_when_fired NUMBER;
BEGIN
    EXECUTE IMMEDIATE
    'UPDATE employee SET salary = :newsal
      WHERE hire_date IS NULL'
    USING no_salary_when_fired;
END;

```

Во-вторых, с помощью функции преобразования типа можно явно преобразовать NULL в типизированное значение:

```

BEGIN
    EXECUTE IMMEDIATE
    'UPDATE employee SET salary = :newsal
      WHERE hire_date IS NULL'
    USING TO_NUMBER (NULL);
END;

```

Работа с объектами и коллекциями

Допустим, нам потребовалось спроектировать систему администрирования корпорации Health\$.Com, работающей в сфере здравоохранения. Для минимизации издержек база данных делается распределенной, поэтому на каждом региональном сервере должны храниться таблицы с информацией о клиентах больниц, принадлежащих Health\$.Com.

Начнем с определения объектного типа (person) и типа VARRAY (preexisting_conditions):

```

/* Файл в Сети: health$.pkg */
CREATE OR REPLACE TYPE person AS OBJECT (
    name VARCHAR2(50), dob DATE, income NUMBER);
/
CREATE OR REPLACE TYPE preexisting_conditions IS TABLE OF VARCHAR2(25);
/

```

После определения этих типов можно создать пакет для управления важнейшей информацией — данными, необходимыми для управления прибылью Health\$.Com. Приведем его спецификацию:

```
PACKAGE health$
AS
    PROCEDURE setup_new_hospital (hosp_name IN VARCHAR2);

    PROCEDURE add_profit_source (
        hosp_name IN VARCHAR2,
        pers IN Person,
        cond IN preexisting_conditions);

    PROCEDURE minimize_risk (
        hosp_name VARCHAR2,
        min_income IN NUMBER := 100000,
        max_preexist_cond IN INTEGER := 0);

    PROCEDURE show_profit_centers (hosp_name VARCHAR2);
END health$;
```

Имея такой пакет, мы можем создать новую таблицу для хранения информации о больнице. Вот как данная задача реализуется в теле пакета:

```
FUNCTION tabname (hosp_name IN VARCHAR2) IS
BEGIN
    RETURN hosp_name || '_profit_center';
END;

PROCEDURE setup_new_hospital (hosp_name IN VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE
        'CREATE TABLE ' || tabname (hosp_name) || ' (
            pers Person,
            cond preexisting_conditions)
            NESTED TABLE cond STORE AS cond_st';
END;
```

Кроме того, пакет позволяет добавлять в таблицу записи о потенциальных клиентах, включая сведения о состоянии их здоровья. Ниже показано, как в теле пакета реализуется эта задача:

```
PROCEDURE add_profit_source (
    hosp_name IN VARCHAR2,
    pers IN Person,
    cond IN preexisting_conditions)
IS
BEGIN
    EXECUTE IMMEDIATE
        'INSERT INTO ' || tabname (hosp_name) ||
        ' VALUES (:revenue_generator, :revenue_inhibitors)'
        USING pers, cond;
END;
```

Работа с объектами и коллекциями полностью прозрачна с точки зрения программиста. Вместо них можно было бы использовать числа и даты — ни синтаксис, ни код от этого не изменятся.

Пакет позволяет удалить из списка всех пациентов, страдающих множественными заболеваниями или имеющих низкие доходы. Эта программа самая сложная:

```
PROCEDURE minimize_risk (
    hosp_name VARCHAR2,
    min_income IN NUMBER := 100000,
    max_preexist_cond IN INTEGER := 1)
```

```

IS
  cv RefCurTyp;
  human Person;
  known_bugs preexisting_conditions;

  v_table VARCHAR2(30) := tabname (hosp_name);
  v_rowid ROWID;
BEGIN
  /* Находим все записи о больных, у которых количество заболеваний
     превышает пороговое значение или доход ниже определенного уровня,
     и удаляем их из таблицы. */
  OPEN cv FOR
    'SELECT ROWID, pers, cond
      FROM ' || v_table || ' alias
     WHERE (SELECT COUNT(*) FROM TABLE (alias.cond))
           > ' ||
           max_preexist_cond ||
           ' OR
           alias.pers.income < ' || min_income;
  LOOP
    FETCH cv INTO v_rowid, human, known_bugs;
    EXIT WHEN cv%NOTFOUND;
    EXECUTE IMMEDIATE
      'DELETE FROM ' || v_table || ' WHERE ROWID = :rid'
      USING v_rowid;
  END LOOP;
  CLOSE cv;
END;
```



Я решил получать ROWID каждого пациента, чтобы упростить идентификацию записей при выполнении DELETE. Было бы очень удобно использовать запрос FOR UPDATE с последующим включением WHERE CURRENT OF cv в команду DELETE, но это невозможно по двум причинам: (1) чтобы на курсорную переменную можно было ссылаться в команде динамического SQL, она должна обладать глобальной доступностью, и (2) курсорные переменные нельзя объявлять в пакетах, потому что они не обладают долгосрочным состоянием. За дополнительной информацией обращайтесь к разделу «Динамический PL/SQL».

Тем не менее в общем случае при выборке строк, которые вы планируете каким-либо образом изменять, желательно использовать конструкцию FOR UPDATE, чтобы предотвратить возможную потерю изменений.

Динамический PL/SQL

Динамический PL/SQL открывает перед программистом ряд интереснейших возможностей. Только подумайте: в то время как пользователь работает с вашим приложением, вы можете с помощью NDS:

- создать программу, в том числе пакет, содержащий глобальные структуры данных;
- получать и модифицировать значения глобальных переменных по именам;
- вызывать функции и процедуры, имена которых не известны во время компиляции.

Я использовал эту технологию для разработки очень гибких генераторов кода, которые динамически формируют вычислительные блоки и множество других компонентов программы. Динамический PL/SQL позволяет работать на более высоком уровне абстракции, но реализовать предоставляемые им возможности одновременно и сложно, и невероятно интересно.

Несколько правил и советов, которые пригодятся вам при работе с динамическими блоками PL/SQL и NDS:

- Динамическая строка должна быть допустимым блоком PL/SQL, начинаться с ключевого слова **DECLARE** или **BEGIN** (которому *может* предшествовать метка или комментарий), а завершаться командой **END** и точкой с запятой (в противном случае она не считается кодом PL/SQL).
- В динамическом блоке доступны только глобальные элементы PL/SQL (функции, процедуры и другие элементы, объявленные в спецификации пакета). Динамические блоки PL/SQL выполняются вне локального внешнего блока.
- Ошибки, возникающие в динамическом блоке PL/SQL, могут быть перехвачены и обработаны локальным блоком, в котором с помощью команды **EXECUTE IMMEDIATE** выполнялись динамические команды SQL.

Построение динамических блоков PL/SQL

Давайте применим эти правила на практике. Сначала напишем маленькую утилиту для выполнения динамического кода PL/SQL:

```
/* Файл в Сети: dynplsql.sp */
PROCEDURE dynPLSQL (blk IN VARCHAR2)
IS
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN ' || RTRIM (blk, ';') || '; END;';
END;
```

В этой программе отражены многие из упоминавшихся выше правил выполнения PL/SQL. Заключение строки в анонимный блок **BEGIN-END** гарантирует, что она будет выполнена как допустимый блок PL/SQL. Например, следующая команда динамически выполняет процедуру `calc_totals`:

```
SQL> exec dynPLSQL ('calc_totals');
```

Используем эту программу для определения того, на какие структуры данных можно ссылаться из динамического блока PL/SQL. В следующем анонимном блоке динамический PL/SQL присваивает значение 5 локальной переменной `num`:

```
<<dynamic>>
DECLARE
    num NUMBER;
BEGIN
    dynPLSQL ('num := 5');
END;
```

Приведенная строка выполняется в собственном блоке **BEGIN-END**, который кажется вложенным в блок `dynamic`. Однако при выполнении этого сценария выводятся следующие сообщения об ошибках:

```
PLS-00201: identifier 'NUM' must be declared
ORA-06512: at "SCOTT.DYNPLSQL", line 4
```

Ядро PL/SQL не может разрешить ссылку на переменную `num`, причем сообщение об ошибке выводится даже при уточнении имени переменной именем блока:

```
<<dynamic>>
DECLARE
    num NUMBER;
BEGIN
    /* Тоже приводит к ошибке PLS-00302! */
    dynPLSQL ('dynamic.num := 5');
END;
```

Предположим, переменная `num` определена в пакете `dynamic`:


```
PACKAGE pkgvars  
IS  
    num NUMBER;  
END pkgvars;
```

Теперь динамическое присваивание этой новой переменной будет выполнено успешно:

```
BEGIN  
    dynPLSQL ('pkgvars.num := 5');  
END;
```

Чем различаются эти два фрагмента данных? В первом случае переменная объявлялась локально в анонимном блоке PL/SQL, а во втором она была объявлена в пакете как общая глобальная переменная. Это различие принципиально для динамического PL/SQL. Оказывается, динамически сконструированный и выполняемый блок PL/SQL не считается вложенным; он выполняется как процедура или функция, вызываемая из текущего блока. Поэтому любые переменные, локальные для текущего или внешнего блока, в динамическом блоке PL/SQL недоступны. Из него можно ссылаться только на глобально определяемые программы и структуры данных, в том числе на отдельные функции и процедуры, а также структуры данных, объявляемые в спецификации пакета.

К счастью, несмотря на это, динамический блок выполняется в контексте вызывающего блока. Это означает, что если в динамическом блоке будет инициировано исключение, оно перехватывается обработчиком исключений вызывающего блока. Поэтому при выполнении в SQL*Plus анонимного блока

```
BEGIN  
    dynPLSQL ('undefined.packagevar := 'abc'');  
EXCEPTION  
    WHEN OTHERS  
    THEN  
        DBMS_OUTPUT.PUT_LINE (SQLCODE);  
END;
```

исключение будет обработано.



Присваивание, выполняемое в этом анонимном блоке, дает пример использования косвенной ссылки: я обращаюсь к переменной не напрямую, а указываю ее имя. Продукт Oracle Forms Builder (ранее известный под названиями SQL*Forms и Oracle Forms) предоставляет реализацию косвенных ссылок в программах NAME_IN и COPY. Эта возможность позволяет разработчикам строить логику, доступную для всех форм в приложении. PL/SQL не поддерживает косвенные ссылки, но их можно реализовать в динамическом PL/SQL. Пример такой реализации приведен в файле dynvar.pkg на сайте книги.

Ниже приведены дополнительные примеры применения динамического PL/SQL. Хочется надеяться, что они вызовут у вас интерес.

Замена повторяющегося кода динамическими блоками

Честное слово, совершенно правдивая история. Когда я консультировал одну страховую компанию в Чикаго, меня попросили что-нибудь сделать с одной крайне неприятной программой. Она была просто огромной и постоянно увеличивалась в размерах — предполагалось, что вскоре она станет настолько большой, что перестанет компилироваться. К моему изумлению, эта программа выглядела так:

```

PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
    IF line = 1 THEN process_line1;
    ELSIF line = 2 THEN process_line2;
    ...
    ELSIF line = 514 THEN process_line514;
    ...
    ELSIF line = 2057 THEN process_line2057;
    END IF;
END;

```

Каждый номер представлял один из пунктов страхового полиса, которые печатаются мелким шрифтом и помогают страховой компании добиться своей главной цели (сокращения страховых выплат). Для каждого номера существовала программа `process_line`, которая обрабатывала все подробности. И по мере того, как страховая компания добавляла новые исключения, программа становилась все больше и больше. Сами понимаете, масштабируемость такого подхода оставляет желать лучшего.

Чтобы избежать подобной путаницы, программист должен обращать внимание на повторяющийся код. Если вам удастся выявить закономерность, вы сможете либо создать универсальную программу, которая эту закономерность инкапсулирует, либо выразить ее в виде динамической конструкции SQL.

В тот раз я решил проблему с использованием `DBMS_SQL`, но динамический SQL стал бы идеальным решением. Реализация NDS выглядела бы так:

```

PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN process_line' || line || '; END;';
END;

```

Тысячи строк кода свелись к одной исполняемой команде! Конечно, в большинстве случаев процесс выявления закономерности и ее преобразования в динамический SQL будет не столь прямолинейным, и все же потенциал для повышения эффективности огромен.

Рекомендации для NDS

К настоящему моменту вы должны уже достаточно хорошо понимать, как «встроенный» динамический SQL работает в PL/SQL. В этом разделе рассматриваются некоторые аспекты, которые следует учитывать при построении коммерческих приложений, использующих эту возможность PL/SQL.

Используйте права вызывающего для совместно используемых программ

Допустим, я создал несколько довольно полезных программ общего назначения, включая функции и процедуры для решения следующих задач:

- Выполнение произвольной команды DDL.
- Получение количества строк в произвольной таблице.
- Получение количества строк для каждой группировки по заданному столбцу.

Все эти средства весьма полезны, и я хочу, чтобы все участники моей группы разработки могли пользоваться ими. Итак, я компилирую их в схеме `COMMON` и предоставляю полномочия `EXECUTE` для этих программ категории `PUBLIC`.

Однако в этой политике кроется одна проблема. Сандра подключается к своей схеме SANDRA и выполняет следующую команду:

```
SQL> EXEC COMMON.exec_DDL ('create table temp (x date)');
```

При этом она непреднамеренно создает таблицу в схеме COMMON — если только я не воспользуюсь моделью прав вызывающего, описанной в главе 24. Модель прав вызывающего означает, что вы определяете свои хранимые программы так, что они выполняются с привилегиями вызывающей, а не определяющей схемы (режим по умолчанию, начиная с Oracle8i, и единственно возможный вариант в предыдущих версиях).

К счастью, новая возможность достаточно проста в использовании. Ниже приведена версия моей процедуры `exec_ddl`, которая выполняет произвольную команду DDL, но всегда использует вызывающую схему:

```
PROCEDURE exec_DDL (ddl_string IN VARCHAR2)
  AUTHID CURRENT_USER
IS
BEGIN
  EXECUTE IMMEDIATE ddl_string;
END;
```

Я рекомендую включать секцию `AUTHID CURRENT_USER` во все ваши динамические программы SQL — и особенно в те, которые вы собираетесь распространять среди других разработчиков.

Прогнозирование и обработка динамических ошибок

Любое надежное приложение должно прогнозировать и обрабатывать ошибки. В динамическом SQL задачи выявления и исправления ошибок оказываются особенно непростыми.

Иногда самым сложным аспектом построения и выполнения динамических программ SQL оказывается построение правильной строки динамического SQL. Вероятно, вам придется объединить список столбцов со списком таблиц и добавить секцию `WHERE`, которая изменяется при каждом выполнении. Вы должны объединить все компоненты посредством конкатенации, правильно расставить запятые, не ошибиться с `AND` и `OR` и т. д. А что произойдет, если вы где-то ошибетесь?

База данных Oracle выдаст ошибку. Обычно эта ошибка сообщает, что именно не так со строкой SQL, но обычно эта информация оставляет желать лучшего. Рассмотрим следующий кошмарный сценарий: я строю самое сложное приложение PL/SQL в мире. В нем сплошь и рядом используется динамический SQL, но это нормально — я же профессионал в NDS. Я за считанные секунды ввожу `EXECUTE IMMEDIATE`, `OPEN FOR` и вообще все нужные команды. В обработке ошибок задействованы стандартные программы обработки ошибок, которые я сам написал и которые выводят сообщение об ошибке при возникновении исключения.

Приходит время протестировать готовое приложение. Я запускаю тестовый сценарий, который выполняет большую часть моего кода; сценарий хранится в файле `testall.sql` (вы найдете его на сайте книги. Начинаем тестирование:

```
SQL> @testall
```

И тут к моему жесточайшему разочарованию на экране появляются следующие сообщения:

```
ORA-00942: table or view does not exist
ORA-00904: invalid column name
ORA-00921: unexpected end of SQL command
ORA-00936: missing expression
```

И что мне с ними делать? К какой команде SQL относится каждое из этих сообщений? Мораль: при интенсивном использовании динамического SQL очень легко запутаться и потратить лишнее время на отладку кода — если только не принять необходимые меры при написании динамического SQL.

Мои рекомендации:

- Всегда включайте раздел обработки ошибок в код, вызывающий EXECUTE IMMEDIATE или OPEN FOR.
- В каждом обработчике сохраняйте и/или выводите сообщения и команды SQL при возникновении ошибок.
- Рассмотрите возможность добавления «трассировки» перед такими командами, чтобы вы могли легко наблюдать за динамическим SQL в процессе построения и выполнения.

Как эти рекомендации отражаются на вашем коде? Давайте применим их к процедуре `exec_ddl`, а затем пойдем по пути обобщения. Отправная точка выглядит так:

```
PROCEDURE exec_ddl (ddl_string IN VARCHAR2)
  AUTHID CURRENT_USER IS
BEGIN
  EXECUTE IMMEDIATE ddl_string;
END;
```

Добавим раздел обработки ошибок, который будет выводить информацию о возникающих проблемах:

```
/* Файл в Сети: execddl.sp */
PROCEDURE exec_ddl (ddl_string IN VARCHAR2)
  AUTHID CURRENT_USER IS
BEGIN
  EXECUTE IMMEDIATE ddl_string;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE (
      'Dynamic SQL Failure: ' || DBMS_UTILITY.FORMAT_ERROR_STACK);
    DBMS_OUTPUT.PUT_LINE (
      '   on statement: "' || ddl_string || '"');
    RAISE;
END;
```

При попытке использования этой версии для создания таблицы с недопустимым синтаксисом происходит следующее:

```
SQL> EXEC execddl ('create table x')
Dynamic SQL Failure: ORA-00906: missing left parenthesis
on statement: "create table x"
```

Конечно, в окончательной версии можно применить более совершенные средства, чем встроенный пакет DBMS_OUTPUT.



Если при использовании DBMS_SQL попытка разбора завершается неудачей и курсор не будет явно закрыт в разделе ошибок, он так и останется открытым, что может привести к возможным ошибкам ORA-01000: maximum open cursors exceeded. В NDS это не случится; курсорные переменные, объявленные в локальной области действия автоматически закрываются при завершении блока (с освобождением занимаемой ими памяти).

А теперь взглянем на происходящее в перспективе: если задуматься, процедура `exec_ddl` не привязана к командам DDL. Она может использоваться для выполнения произвольной строки SQL, не требующей использования секций USING или INTO. С этой точки

зрения можно создать одну программу, которая может использоваться вместо прямого вызова `EXECUTE IMMEDIATE`; со встроенной обработкой ошибок. Я включаю такую процедуру в пакет `ndsutil`.

Я мог бы даже создать аналогичную программу для `OPEN FOR` — снова только для ситуаций, не требующих секции `USING`. Поскольку `OPEN FOR` задает значение курсора, вероятно, реализацию стоит оформить в виде функции, возвращающей тип слабой ссылки `REF CURSOR`.

В итоге получается пакетная реализация следующего вида:

```
PACKAGE ndsutil
IS
    FUNCTION openFor (sql_string IN VARCHAR2) RETURN SYS_REFCURSOR;
END;
```

Пакет содержит полную реализацию функции; ее тело практически не отличается от приведенной ранее процедуры `exec_d11`.

Параметры вместо конкатенации

В большинстве случаев можно использовать два разных механизма вставки программных значений в строку SQL: передачу параметров и конкатенацию. В следующей таблице приведены примеры применения обоих подходов к построению динамической команды `UPDATE`.

| Параметры | Конкатенация |
|--|---|
| <code>EXECUTE IMMEDIATE 'UPDATE ' tab 'SET sal = :new_sal' USING v_sal;</code> | <code>EXECUTE IMMEDIATE 'UPDATE ' tab 'SET sal = ' v_sal;</code> |

Первый способ подразумевает использование формальных параметров и секции `USING`; конкатенация упрощает этот процесс до прямого включения значений в строку SQL. Когда следует применять каждый из этих вариантов? Я рекомендую применять передачу параметров там, где это возможно (см. следующий раздел). Для этого есть четыре причины:

- **Передача параметров обычно работает быстрее.** При передаче значения в параметре строка SQL не содержит значения — только имя формального параметра. Следовательно, вы можете связать разные значения с одной командой SQL без ее изменения. Так как команда SQL при этом не изменяется, повышается вероятность того, что ваше приложение воспользуется заранее разобранными курсорами, кэшированными в области SGA базы данных.

Примечание 1. Я говорю о «повышении вероятности», потому что в мире оптимизации Oracle есть очень мало абсолютных истин. Например, один из возможных недостатков передачи параметров заключается в том, что оптимизатор, основанный на определении затрат, располагает меньшей информацией и с меньшей вероятностью построит лучший план исполнения команды SQL.

Примечание 2. Если вам понадобится несколько раз выполнить одну команду SQL с разными параметрами, подумайте об использовании `DBMS_SQL` — возможно, вам удастся полностью избежать разбора команды, что невозможно с NDS. (См. «Минимальный разбор динамических курсоров».)

- **Передача параметров упрощает написание и сопровождение кода.** При передаче параметров не нужно беспокоиться о преобразовании типа данных; все делается автоматически ядром NDS. Собственно, передача параметров сводит преобразования к минимуму, потому что она работает со встроенными типами данных. При использовании конкатенации часто приходится писать очень сложные строковые выражения с длинными сериями одиночных кавычек, вызовами функций `TO_DATE` и `TO_CHAR` и т. д.

- **Передача параметров помогает избежать неявных преобразований.** С конкатенацией может оказаться, что вы случайно поручите базе данных выполнение неявных преобразований. В некоторых обстоятельствах база данных может выполнять не те преобразования, которые вам нужны; это может помешать использованию индексов.
- **Передача параметров сводит к минимуму опасность внедрения кода.** Одна из самых серьезных опасностей динамического SQL заключается в том, что вы пишете очень обобщенный код, который должен использоваться определенным образом, однако в зависимости от данных, передаваемых пользователем, полученная динамическая команда будет выполнять совершенно другие операции. Иначе говоря, пользователь может «внедрить» в команду SQL нежелательные операции. Пример приведен в следующем разделе.

Впрочем, у передачи параметров есть и потенциальные недостатки. В версиях, предшествующих 11.1, она нарушает использование гистограммной статистики, потому что значения параметров присваиваются только после разбора команды. Оптимизатор, основанный на затратах, может располагать меньшим объемом информации, а это мешает ему построить оптимальный план выполнения команд SQL. В версии 11.1 и выше эта статистика используется функцией адаптивного совместного использования курсоров.

Вероятно, разработчик PL/SQL должен стремиться прежде всего к написанию «чистого», понятного и простого в сопровождении кода. При многочисленных операциях конкатенации появляются команды, которые выглядят примерно так:

```
EXECUTE IMMEDIATE
  'UPDATE employee SET salary = ' || val_in ||
  ' WHERE hire_date BETWEEN ' ||
  '   TO_DATE (''' || TO_CHAR (v_start) || ''') ' ||
  ' AND ' ||
  '   TO_DATE (''' || TO_CHAR (v_end) || ''')';
```

С переходом на передачу параметров код становится более понятным:

```
EXECUTE IMMEDIATE
  'UPDATE employee SET salary = :val
  WHERE hire_date BETWEEN :lodate AND :hodate'
  USING v_sal, v_start, v_end;
```

И хотя в некоторых ситуациях конкатенация работает более эффективно, не беспокойтесь об этом, пока вы или ваш администратор базы данных не определит, что проблемы возникли из-за конкретной команды динамического SQL. Иначе говоря, переходите с передачи параметров на конкатенацию только при выявлении «узких мест» в программе — и это должно быть исключением, а не правилом.

Минимизация опасности внедрения кода

Многие веб-приложения предоставляют исключительно гибкие возможности для конечного пользователя. Гибкость часто достигается применением динамических блоков PL/SQL и SQL. Рассмотрим пример очень типичной процедуры получения строк:

```
/* Файл в Сети: code_injection.sql */
PROCEDURE get_rows (
  table_in IN VARCHAR2, where_in IN VARCHAR2
)
IS
BEGIN
  EXECUTE IMMEDIATE
    'DECLARE
     l_row ' || table_in || '%ROWTYPE;
  BEGIN
    SELECT * INTO l_row
```

```
FROM ' || table_in || ' WHERE ' || where_in || ' ;  
END;';  
END get_rows;
```

Выглядит совершенно невинно, но в действительности этот код открывает зияющие бреши в защите приложения. Рассмотрим следующий блок:

```
BEGIN  
  get_rows ('EMPLOYEE'  
    , 'employee_id=7369;  
    EXECUTE IMMEDIATE  
      'CREATE PROCEDURE backdoor (str VARCHAR2)  
        AS BEGIN EXECUTE IMMEDIATE str; END;''' );  
END;  
/
```

После выполнения этого кода создается процедура, которая выполнит любую команду, переданную в динамической строке. Например, я могу использовать UTL_FILE для получения содержимого любого файла в системе, а затем создать (и удалить) любую таблицу или объект — эта возможность будет ограничена только привилегиями, определенными для схемы владельца.

Внедрение кода, также называемое внедрением SQL, способно серьезно повредить безопасности любого приложения. Выполнение динамических блоков PL/SQL открывает больше всего возможностей для внедрения. Хотя внедрение — очень большая тема, которую невозможно полноценно рассмотреть в этой теме, я приведу несколько рекомендаций, которые снизят вероятность внедрения в ваших приложениях. Дополнительные советы по безопасности приводятся в главе 23.

Жесткое ограничение привилегий для пользовательских схем

Лучший способ свести к минимуму риск внедрения — позаботиться о том, чтобы привилегии любой схемы, к которой подключается внешний пользователь, были жестко ограничены.

Запретите таким схемам создание объектов базы данных, удаление объектов базы данных или прямой доступ к таблицам. Запретите выполнение пакетов, которые взаимодействуют (или могут использоваться для взаимодействия) с операционной системой — таких, как UTL_SMTP, UTL_FILE, UTL_TCP (и сопутствующих пакетов), а также DBMS_PIPE.

Такая схема должна обладать привилегиями только для выполнения хранимых программ, определенных в *другой* схеме. Такой код PL/SQL тщательно планируется таким образом, чтобы он мог выполнять только ограниченный набор операций. При определении программ в этих исполняемых схемах, использующих динамический SQL, обязательно определите подпрограммы с конструкцией AUTHID CURRENT_USER. В этом случае все программы SQL будут выполняться с ограниченными привилегиями схемы текущего подключения.

Используйте передачу параметров там, где это возможно

Строгое соблюдение правил использования параметров в сочетании со встроенными средствами анализа и автоматизированным отклонением потенциально опасных строк позволит свести к минимуму опасность внедрения кода.

Правда, передача параметров оборачивается некоторой потерей гибкости. Например, в процедуре `get_rows` мне пришлось заменить полностью динамическую секцию `WHERE` менее универсальной конструкцией, которая лучше соответствует ожидаемому поведению приложения. Пример слегка видоизмененной процедуры `get_rows`:

```

PROCEDURE get_rows (
    table_in IN VARCHAR2, value1_in in VARCHAR2, value2_in IN DATE
)
IS
    l_where VARCHAR2(32767);
BEGIN
    IF table_in = 'EMPLOYEES'
    THEN
        l_where := 'last_name = :name AND hire_date < :hdate';
    ELSIF table_in = 'DEPARTMENTS'
    THEN
        l_where := 'name LIKE :name AND incorporation_date = :hdate';
    ELSE
        RAISE_APPLICATION_ERROR (
            -20000, 'Invalid table name for get_rows: ' || table_in);
    END IF;
    EXECUTE IMMEDIATE
        'DECLARE l_row ' || table_in || '%ROWTYPE;
        BEGIN
            SELECT * INTO l_row
            FROM ' || table_in || ' WHERE ' || l_where || ';
        END;'
        USING value1_in, value2_in;
END get_rows;
/

```

В этой версии секция WHERE использует два параметра; у злоумышленника не будет возможности выполнить постороннюю операцию из-за конкатенации. Я также проверяю имя таблицы и убеждаюсь в том, что оно соответствует ожидаемому. Это помогает избежать вызовов функций в секции FROM (так называемые *табличные функции*), которые тоже могут привести к нежелательным последствиям.

Проверка динамического текста

У рекомендаций из предыдущих разделов есть один недостаток: они зависят от сознательности конкретного разработчика или администратора. Эти рекомендации станут неплохой отправной точкой, и все же их недостаточно. Вероятно, в программу стоит включить проверки, которые убедятся в том, что пользовательский текст не содержит «опасных» символов — например, точки с запятой.

Я написал программу SQL Guard, которая идет по другому пути: она анализирует строку, введенную пользователем, и определяет, создает ли она риск внедрения SQL. Затем программист может решить, стоит ли выполнять эту команду (и возможно, зарегистрировать подозрительный текст в журнале). Код и руководство пользователя SQL Guard находятся в файле `sqlguard.zip` на сайте книги.

В программе SQL Guard проверки, по которым определяется риск внедрения SQL, могут настраиваться пользователем. Другими словами, SQL Guard поставляется с набором предопределенных проверок; вы можете удалять и добавлять проверки в список для схем внедрения SQL, специфических для рабочей среды вашего приложения.

Невозможно создать механизм, который будет со 100% вероятностью перехватывать любые возможные попытки внедрения SQL. Тем не менее если вы решите использовать SQL Guard, эта программа (на мой взгляд) поможет вам:

- донести до ваших разработчиков необходимость учитывать угрозу внедрения SQL;
- предотвратить наиболее распространенные атаки внедрения SQL;
- проанализировать кодовую базу для выявления возможных путей внедрения.

Использование DBMS_ASSERT для проверки входных данных

Пакет DBMS_ASSERT проверяет действительность вводимого пользователем имени объекта SQL (например, имени схемы или таблицы). Пакет DBMS_ASSERT впервые был документирован в Oracle Database 11g. С того времени он был адаптирован для следующих версий Oracle: 8.1, 9.2, 10.1 и 10.2. В некоторых случаях он доступен в составе обновлений (в том числе и критических). Возможно, перед началом использования пакета вам придется связаться со службой поддержки Oracle.

DBMS_ASSERT.SIMPLE_SQL_NAME получает строку, которая должна содержать имя объекта SQL. Если имя проходит проверку, то функция возвращает строку без изменения. Если же имя недействительно, Oracle инициирует исключение DBMS_ASSERT.INVALID_SQL_NAME.

За более подробным изложением темы обращайтесь к статье «How to write SQL injection proof PL/SQL», размещенной в Oracle Technology Network.

Когда следует использовать DBMS_SQL

Если перед вами станет вопрос, какую из описанных технологий следует выбрать, начать следует со встроенного динамического SQL (вместо DBMS_SQL), поскольку он гораздо проще в применении, а программный код получается более коротким и обычно содержит меньшее количество ошибок. Кроме того, такой код проще в сопровождении, а обычно и выполняется более эффективно.

И все же в некоторых ситуациях приходится использовать пакет DBMS_SQL. Эти ситуации описаны ниже.

Получение информации о столбцах запроса

Пакет DBMS_SQL позволяет описывать столбцы динамического курсора, возвращая информацию о каждом столбце в ассоциативном массиве записей. Перед разработчиком открывается возможность написания универсального кода работы с курсорами; она особенно полезна, если вы пишете динамический SQL категории 4 и не уверены, сколько именно столбцов задействовано в выборке.

При вызове этой программы необходимо объявить коллекцию PL/SQL на базе типа коллекции DBMS_SQL.DESC_TAB (или DESC_TAB2, если запрос может возвращать имена столбцов, длина которых превышает 30 символов). После этого вы можете использовать методы коллекций для перебора таблицы и извлечения информации о курсоре. Следующий анонимный блок демонстрирует основные действия, выполняемые при работе с пакетом:

```
DECLARE
  cur PLS_INTEGER := DBMS_SQL.OPEN_CURSOR;
  cols DBMS_SQL.DESC_TAB;
  ncols PLS_INTEGER;
BEGIN
  -- Разбор запроса
  DBMS_SQL.PARSE
    (cur, 'SELECT hire_date, salary FROM employees', DBMS_SQL.NATIVE);
  -- Получение информации о столбцах
  DBMS_SQL.DESCRIBE_COLUMNS (cur, ncols, cols);
  -- Вывод каждого из имен столбцов
  FOR colind IN 1 .. ncols
  LOOP
    DBMS_OUTPUT.PUT_LINE (cols (colind).col_name);
  END LOOP;
  DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

Чтобы упростить использование DESCRIBE_COLUMNS, я создал пакет, который скрывает большинство технических подробностей и упрощает использование этой функциональности. Спецификация пакета выглядит так:

```
/* Файл в Сети: desccols.pkg */
PACKAGE desccols
IS
    FUNCTION for_query (sql_in IN VARCHAR2)
        RETURN DBMS_SQL.desc_tab;
    FUNCTION for_cursor (cur IN PLS_INTEGER)
        RETURN DBMS_SQL.desc_tab;
    PROCEDURE show_columns (
        col_list_in IN DBMS_SQL.desc_tab
    );
END desccols;
```

Функция `for_query` используется в том случае, если вы хотите получить информацию о столбцах динамического запроса, но в остальном не собираетесь использовать `DBMS_SQL`.

Использование пакета продемонстрировано в следующем сценарии:

```
/* Файл в Сети: desccols.sql */
DECLARE
    cur    INTEGER           := DBMS_SQL.open_cursor;
    tab    DBMS_SQL.desc_tab;
BEGIN
    DBMS_SQL.parse (cur
        , 'SELECT last_name, salary, hiredate FROM employees'
        , DBMS_SQL.native
    );
    tab := desccols.for_cursor (cur);
    desccols.show (tab);
    DBMS_SQL.close_cursor (cur);
    --
    tab := desccols.for_query ('SELECT * FROM employees');
    desccols.show (tab);
END;
```

Поддержка требований категории 4

Пакет `DBMS_SQL` поддерживает динамический SQL категории 4 (переменное количество столбцов или передаваемых параметров) более естественно, чем NDS. Вы уже видели, что для реализации категории 4 в NDS необходимо перейти на динамический PL/SQL, а многие разработчики обычно стараются обойтись без этого высокого уровня абстракции.

Когда вы сталкиваетесь с категорией 4? Конечно, при построении интерфейсной части для поддержки несистематизированных запросов, генерируемых пользователями, или при построении обобщенной программы построения отчетов, которая динамически строит формат отчета и содержимое во время выполнения. Разберем одну из вариаций на эту тему: построение процедуры PL/SQL для вывода содержимого таблицы — *любой* таблицы, заданной пользователем во время выполнения. Здесь мы ограничимся рассмотрением аспектов, относящихся непосредственно к динамическому SQL; за полной реализацией обращайтесь к файлу `intab.sp` на сайте книги.

Процедурный интерфейс

В своей реализации я буду использовать PL/SQL и `DBMS_SQL`. Но прежде чем браться за написание кода, необходимо создать спецификацию. Как будет вызываться эта процедура? Какую информацию должен предоставить пользователь (в данном случае

разработчик)? Что должен ввести пользователь для получения желаемого вывода? В следующей таблице перечислены входные данные моей процедуры `intab` (сокращение от «in table»).

| Параметр | Описание |
|----------------------|---|
| Имя таблицы | Обязательно (главная информация, используемая программой) |
| Секция WHERE | Не обязательно. Позволяет ограничить строки, возвращаемые запрос. Если условие не задано, возвращаются все строки. Параметр также может использоваться для передачи условий ORDER BY и HAVING, следующих непосредственно за WHERE |
| Фильтр имен столбцов | Не обязательно. Если вы не хотите выводить все столбцы таблицы, передайте список, разделенный запятыми; в этом случае будут использоваться только указанные вами столбцы |

С этими данными спецификация моей процедуры принимает следующий вид:

```
PROCEDURE intab (
    table_in IN VARCHAR2
    , where_in IN VARCHAR2 DEFAULT NULL
    , colname_like_in IN VARCHAR2 := '%'
);
```

Ниже приводятся примеры вызовов `intab` с результатами. Начнем с вывода всего содержимого таблицы `emp`:

```
SQL> EXEC intab ('emp');
```

| Contents of emp | | | | | | | |
|-----------------|--------|-----------|------|----------|--------|------|--------|
| EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO |
| 7369 | SMITH | CLERK | 7902 | 12/17/80 | 120000 | 800 | 20 |
| 7499 | ALLEN | SALESMAN | 7698 | 02/20/81 | 120000 | 1600 | 30 |
| 7521 | WARD | SALESMAN | 7698 | 02/22/81 | 120000 | 1250 | 30 |
| 7566 | JONES | MANAGER | 7839 | 04/02/81 | 120000 | 2975 | 20 |
| 7654 | MARTIN | SALESMAN | 7698 | 09/28/81 | 120000 | 1250 | 30 |
| 7698 | BLAKE | MANAGER | 7839 | 05/01/81 | 120000 | 2850 | 30 |
| 7782 | CLARK | MANAGER | 7839 | 06/09/81 | 120000 | 2450 | 10 |
| 7788 | SCOTT | ANALYST | 7566 | 04/19/87 | 120000 | 3000 | 20 |
| 7839 | KING | PRESIDENT | | 11/17/81 | 120000 | 5000 | 10 |
| 7844 | TURNER | SALESMAN | 7698 | 09/08/81 | 120000 | 1500 | 30 |
| 7876 | ADAMS | CLERK | 7788 | 05/23/87 | 120000 | 1100 | 20 |
| 7900 | JAMES | CLERK | 7698 | 12/03/81 | 120000 | 950 | 30 |
| 7902 | FORD | ANALYST | 7566 | 12/03/81 | 120000 | 3000 | 20 |

А теперь перейдем к работникам отдела 10, ограничивая максимальную длину текстовых столбцов 20 символами:

```
SQL> EXEC intab ('emp', 'deptno = 10 ORDER BY sal');
```

| Contents of emp | | | | | | | |
|-----------------|--------|-----------|------|----------|--------|------|--------|
| EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO |
| 7934 | MILLER | CLERK | 7782 | 01/23/82 | 120000 | 1300 | 10 |
| 7782 | CLARK | MANAGER | 7839 | 06/09/81 | 120000 | 2450 | 10 |
| 7839 | KING | PRESIDENT | | 11/17/81 | 120000 | 5000 | 10 |

Переходим к совершенно другой таблице с другим количеством столбцов:

```
SQL> EXEC intab ('dept')
```

| Contents of dept | | |
|------------------|-------|-----|
| DEPTNO | DNAME | LOC |

| | | |
|----|------------|----------|
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |

Обратите внимание: пользователю не приходится передавать информацию о структуре таблицы. Моя программа получает эту информацию самостоятельно — собственно, именно эта особенность процедуры `intab` делает ее примером динамического SQL категории 4.

Последовательность построения `intab`

Вывод содержимого произвольной таблицы происходит по следующей схеме:

1. Построение и разбор команды `SELECT` (с использованием `OPEN_CURSOR` и `PARSE`).
2. Связывание всех локальных переменных с их «заместителями» в запросе (с использованием `BIND_VARIABLE`).
3. Определение каждого курсора для этого запроса (с использованием `DEFINE_COLUMN`).
4. Выполнение и выборка строк базы данных (с использованием `EXECUTE` и `FETCH_ROWS`).
5. Получение значений из выбранной строки таблицы и их включение в выводимый текст (с использованием `COLUMN_VALUE`). Затем строка выводится выводом процедуры `PUT_LINE` пакета `DBMS_OUTPUT`.



В настоящее время моя реализация `intab` не поддерживает привязку параметров в аргументе `where_in`, поэтому мы не будем подробно рассматривать код шага 2.

Построение `SELECT`

Чтобы извлечь данные из таблицы, необходимо построить команду `SELECT`. Структура этого запроса определяется различными входными данными процедуры (имя таблицы, условие `WHERE` и т. д.) и содержимым словаря данных. Помните, что пользователь не должен предоставлять список столбцов — вместо этого я должен идентифицировать столбцы таблицы и извлечь список из представления словаря данных. Я решил использовать представление `ALL_TAB_COLUMNS` процедуры `intab`, чтобы пользователь мог просматривать содержимое не только таблиц, владельцем которых он является (доступных в `USER_TAB_COLUMNS`), но и любых таблиц, для которых он обладает доступом `SELECT`.

Для получения информации о столбцах таблицы используется следующий курсор:

```
CURSOR col_cur
  (owner_in IN VARCHAR2,
   table_in IN VARCHAR2)
IS
  SELECT column_name, data_type,
         data_length,
         data_precision, data_scale
  FROM all_tab_columns
 WHERE owner = owner_in
        AND table_name = table_in;
```

С этим курсором я получаю имя, тип данных и длину для каждого столбца таблицы. Как сохранить всю эту информацию в моей программе PL/SQL?

- Чтобы ответить на этот вопрос, необходимо подумать над тем, как эти данные будут использоваться. Как выясняется, у них есть много разных вариантов использования — например:

- Чтобы построить список выборки для запроса, я буду использовать имена столбцов.
- Чтобы вывести содержимое таблицы в удобном виде, необходимо вывести над данными заголовки столбцов, которые должны быть размещены по горизонтали в правильных позициях. Следовательно, для каждого столбца необходимо знать имя и длину данных.
- Чтобы осуществить выборку данных в динамическом курсоре, необходимо определить столбцы курсора вызовами `DEFINE_COLUMN`. Для этого нужно знать типы и длины данных.
- Чтобы извлечь данные из выбранной строки с использованием `COLUMN_VALUE`, я должен знать типы данных всех столбцов, а также их количество.
- Чтобы вывести данные, я должен построить строку со всем данными (преобразуя числа и даты функцией `TO_CHAR`). Данные необходимо дополнить по ширине для размещения под заголовками столбцов, как это было сделано в строке заголовка.

Моя программа многократно использует информацию столбцов, однако повторных обращений к словарию данных хотелось бы избежать. Соответственно, при получении данных столбцов из представления `ALL_TAB_COLUMNS` я сохраню эти данные в трех разных коллекциях PL/SQL, описанных в следующей таблице.

| Коллекция | Описание |
|-----------|--|
| colname | Имена столбцов |
| coltype | Типы данных столбцов (строки с описаниями типов) |
| collen | Количество символов, необходимых для вывода данных столбца |

Таким образом, для третьего столбца таблицы `emp` элемент `colname(3)` содержит строку «SAL», элемент `coltype(3)` содержит строку «NUMBER», а элемент `collen(3)` равен 7 и т. д.

Информация имени и типа данных берется прямо из словаря данных. С вычислением длины столбца дело обстоит сложнее, но для написания динамического SQL категории 4 это не столь существенно. При желании читатель может изучить файл самостоятельно.

Вся логика применяется в цикле `FOR` с курсором, который перебирает все столбцы таблицы (в соответствии с определением `ALL_COLUMNS`). Этот цикл (приведенный в следующем примере) заполняет мою коллекцию PL/SQL:

```
FOR col_rec IN col_cur (owner_nm, table_nm)
LOOP
    /* Построение списка выборки для запроса. */
    col_list := col_list || ', ' || col_rec.column_name;
    /* Сохранение типа данных и длины для вызовов DEFINE_COLUMN. */
    col_count := col_count + 1;
    colname (col_count) := col_rec.column_name;
    coltype (col_count) := col_rec.data_type;

    /* Построение строки заголовка. */
    col_header :=
        col_header || ' ' || RPAD (col_rec.column_name, v_length);
END LOOP;
```

После завершения цикла список выборки построен, коллекции PL/SQL заполнены информацией о столбцах, необходимой для вызовов `DBMS_SQL.DEFINE_COLUMN` и `DBMS_SQL.COLUMN_VALUE`, а также создана строка заголовка. Неплохо для одного цикла!

Теперь нужно разобрать запрос и построить разные столбцы объекта динамического курсора.

Определение структуры курсора

Фаза разбора достаточно тривиальна. Я просто собираю команду SQL из обработанных компонентов — прежде всего, из только что построенного списка столбцов (переменная `col_list`):

```
DBMS_SQL.PARSE
(cur,
 'SELECT ' || col_list ||
 ' FROM ' || table_in || ' ' || where_clause,
 DBMS_SQL.NATIVE);
```

Я не собираюсь ограничиться разбором — курсор нужно выполнить. Но перед этим необходимо его структурировать. В `DBMS_SQL` при открытии курсора вы всего лишь получаете дескриптор для блока памяти. При разборе команды SQL эта команда связывается с упомянутой памятью. Однако необходимо сделать следующий шаг — определить столбцы в курсоре, чтобы он мог использоваться для хранения полученных данных.

Для динамического SQL категории 4 этот процесс достаточно сложен. Я не могу жестко запрограммировать ни количество, ни типы вызовов `DBMS_SQL.DEFINE_COLUMN` в своей программе; часть информации появляется лишь на стадии выполнения. К счастью, в случае `intab` имеются коллекции с данными обо всех столбцах. Остается лишь создать вызов `DBMS_SQL.DEFINE_COLUMN` для каждого столбца, определенного в коллекции `colname`. Но прежде чем переходить к коду, стоит сказать пару слов о `DBMS_SQL.DEFINE_COLUMN`.

Заголовок этой встроенной процедуры выглядит так:

```
PROCEDURE DBMS_SQL.DEFINE_COLUMN
(cursor_handle IN INTEGER,
 position IN INTEGER,
 datatype_in IN DATE|NUMBER|VARCHAR2)
```

При ее использовании следует учитывать три обстоятельства:

- Во втором аргументе передается число. `DBMS_SQL.DEFINE_COLUMN` работает не с именами столбцов, а с последовательными позициями столбцов в списке.
- Третий аргумент задает тип данных столбца курсора. В нем передается выражение соответствующего типа. Иначе говоря, `DBMS_SQL.DEFINE_COLUMN` передается не строка (скажем, «`VARCHAR2`»), а переменная, определенная с типом `VARCHAR2`.
- При определении столбца символьного типа необходимо задать максимальную длину значений, которые могут загружаться в курсор.

В контексте процедуры `intab` строка коллекции соответствует N-й позиции списка столбцов. Тип данных хранится в коллекции `coltype`, но его необходимо преобразовать в вызов `DBMS_SQL.DEFINE_COLUMN` с использованием соответствующей локальной переменной. Все это происходит в следующем цикле `FOR`:

```
FOR col_ind IN 1 .. col_count
LOOP
  IF is_string (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN
      (cur, col_ind, string_value, collen (col_ind));

  ELSIF is_number (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN (cur, col_ind, number_value);

  ELSIF is_date (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN (cur, col_ind, date_value);
  END IF;
END LOOP;
```

После завершения цикла будут выполнены вызовы `DEFINE_COLUMN` для каждого столбца, определенного в коллекциях. (В моей версии это все столбцы таблицы. В своей усовершенствованной реализации вы можете ограничиться подмножеством этих столбцов.) Затем я выполняю курсор и перехожу к выборке записей. Фаза исполнения для категории 4 не отличается от других, более простых категорий. Используйте вызов вида:

```
fdbk := DBMS_SQL.EXECUTE (cur);
```

где `fdbk` — объект обратной связи, возвращаемый при вызове `EXECUTE`.

Остается последний шаг: выборка данных и их форматирование для вывода.

Выборка и отображение данных

Я использую цикл `FOR` с курсором для получения каждой строки данных, идентифицированной моим динамическим курсором. Если текущей является первая строка, я вывожу заголовок (тем самым предотвращается вывод заголовка для запроса, не возвращающего данных). Для каждой полученной строки данных я генерирую строку результата и вывожу ее:

```
LOOP
    fdbk := DBMS_SQL.FETCH_ROWS (cur);
    EXIT WHEN fdbk = 0;
    IF DBMS_SQL.LAST_ROW_COUNT = 1
    THEN
        /* Здесь выводится информация заголовка */
        ...
    END IF;

    /* Построение текста с информацией столбцов */
    ...
    DBMS_OUTPUT.PUT_LINE (col_line);
END LOOP;
```

Текст с выводимыми значениями строится в цикле `FOR`. Встроенная функция `DBMS_SQL.COLUMN_VALUE` вызывается для каждого столбца в таблице (информация о котором хранится в моих коллекциях). Как видно из листинга, я использую функции `is_*` для определения типа столбца, а следовательно, и выбора переменной, которая получит значение.

После преобразования значения в строку (необходимого для дат и чисел) я дополняю его справа пробелами до нужной длины (хранящейся в коллекции `collen`), чтобы значение было выровнено по заголовкам столбцов:

```
col_line := NULL;
FOR col_ind IN 1 .. col_count
LOOP
    IF is_string (col_ind)
    THEN
        DBMS_SQL.COLUMN_VALUE (cur, col_ind, string_value);

    ELSIF is_number (col_ind)
    THEN
        DBMS_SQL.COLUMN_VALUE (cur, col_ind, number_value);
        string_value := TO_CHAR (number_value);

    ELSIF is_date (col_ind)
    THEN
        DBMS_SQL.COLUMN_VALUE (cur, col_ind, date_value);
        string_value := TO_CHAR (date_value, date_format_in);
    END IF;
```

продолжение ➤

```

/* Значение дополняется пробелами в строке
   под заголовками столбцов. */
col_line :=
  col_line || ' ' ||
  RPAD (NVL (string_value, ' '), collen (col_ind));
END LOOP;

```

Итак, в вашем распоряжении появляется универсальная процедура для вывода содержимого таблиц базы данных из программ PL/SQL. И снова за подробностями обращайтесь к файлу `intab.sp`; файл `intab_dbms_sql.sp` содержит версию этой процедуры, адаптированную для новых возможностей баз данных и более полно документированную.

Минимальный разбор динамических курсоров

Один из недостатков `EXECUTE IMMEDIATE` заключается в том, что при каждом выполнении динамическая строка обрабатывается заново, что обычно подразумевает разбор, оптимизацию и построение плана выполнения. Для большинства ситуаций, требующих применения динамического SQL, затраты на эти действия компенсируются другими преимуществами NDS (в частности, предотвращением вызовов PL/SQL API, необходимых для `DBMS_SQL`). Однако в некоторых случаях разбор обходится слишком дорого, и тогда `DBMS_SQL` может оказаться лучшим решением.

При использовании `DBMS_SQL` можно явно обойти фазу разбора, если вы знаете, что в динамически исполняемой строке SQL изменяются только параметры. Для этого достаточно не вызывать `DBMS_SQL.PARSE` заново, а ограничиться простой повторной подстановкой значений переменных вызовами `DBMS_SQL.BIND_VARIABLE`. Начнем с очень простого примера, демонстрирующего применение конкретных вызовов пакета `DBMS_SQL`.

Следующий анонимный блок исполняет динамический запрос в цикле:

```

1 DECLARE
2   l_cursor   pls_INTEGER;
3   l_result   pls_INTEGER;
4 BEGIN
5   FOR i IN 1 .. counter
6   LOOP
7     l_cursor := DBMS_SQL.open_cursor;
8     DBMS_SQL.parse
9       (l_cursor, 'SELECT ... where col = ' || i , DBMS_SQL.native);
10    l_result := DBMS_SQL.EXECUTE (l_cursor);
11    DBMS_SQL.close_cursor (l_cursor);
12 END LOOP;
13 END;

```

Действия, выполняемые в цикле, перечислены в следующей таблице.

| Строки | Описание |
|--------|--|
| 7 | Получение курсора — обычного указателя на память, используемую <code>DBMS_SQL</code> |
| 8–9 | Разбор динамического курсора после присоединения единственного изменяемого элемента (переменная <code>i</code>) |
| 10 | Выполнение запроса |
| 11 | Закрытие курсора |

Все происходящее вполне допустимо (и конечно, обычно за выполнением запроса следуют операции выборки), однако в целом `DBMS_SQL` используется неверно. Рассмотрим следующую модификацию тех же действий:

```

DECLARE
  l_cursor   PLS_INTEGER;
  l_result   PLS_INTEGER;

```



```

BEGIN
  l_cursor := DBMS_SQL.open_cursor;
  DBMS_SQL.parse (l_cursor, 'SELECT ... WHERE col = :value'
    , DBMS_SQL.native);

  FOR i IN 1 .. counter
  LOOP
    DBMS_SQL.bind_variable (l_cursor, 'value', i);
    l_result := DBMS_SQL.EXECUTE (l_cursor);
  END LOOP;
  DBMS_SQL.close_cursor (l_cursor);
END;
```

В этом варианте использования DBMS_SQL я объявляю курсор только один раз, потому что могу использовать его с каждым вызовом DBMS_SQL.PARSE. Я также перемещаю вызов разбора за пределы курсора. Поскольку структура команды SQL остается неизменной, курсор не нужно разбирать заново для каждого значения *i*. Итак, я выполняю разбор только один раз, после чего в цикле подставляю в курсор новое значение переменной и выполняю курсор. Когда все будет сделано (после завершения цикла), курсор закрывается.

Возможность явного и раздельного выполнения каждого шага предоставляет разработчику невероятную гибкость (а заодно создает проблемы со сложностью кода DBMS_SQL). Если вы стремитесь именно к этому, с DBMS_SQL трудно соперничать.

Если вы используете DBMS_SQL в своих приложениях, я рекомендую воспользоваться пакетом из файла `dynalloc.pkg` на сайте книги. Этот пакет помогает:

- свести к минимуму операции выделения памяти курсоров посредством повторного использования;
- выполнять осмысленную обработку ошибок для всех операций разбора DBMS_SQL;
- избегать ошибок, связанных с попытками открытия или закрытия уже открытых или закрытых курсоров.

Новые возможности Oracle11g

В Oracle11g появились средства взаимодействия между встроенным динамическим SQL и DBMS_SQL: теперь вы можете пользоваться преимуществами обеих технологий, чтобы совместить оптимальное быстродействие с простейшей реализацией. А если говорить конкретно, появилась возможность преобразования курсоров DBMS_SQL в курсорные переменные, и наоборот.

Функция DBMS_SQL.TO_REFCURSOR

Функция DBMS_SQL.TO_REFCURSOR преобразует *номер* курсора (полученный вызовом DBMS_SQL.OPEN_CURSOR) в слаботипизированную курсорную переменную (объявленную с типом SYS_REFCURSOR или слабым пользовательским типом REF CURSOR). Далее вы можете извлекать данные из курсорной переменной в локальные переменные или даже передать курсорную переменную во внешнюю среду для выборки данных.

Прежде чем передавать курсор SQL функции DBMS_SQL.TO_REFCURSOR, необходимо вызвать для него OPEN, PARSE и EXECUTE; в противном случае произойдет ошибка. После того как курсор будет преобразован, вы уже не сможете работать с ним средствами DBMS_SQL, в том числе и закрывать его. Все операции должны выполняться только через курсорную переменную.

Зачем нужна эта функция? Как упоминалось ранее, DBMS_SQL иногда оказывается предпочтительным или единственным решением для некоторых операций динамического SQL. Допустим, мы знаем, какие конкретные столбцы следует выбрать, но предложение WHERE

запроса содержит неизвестное (на стадии компиляции) количество формальных параметров, что не позволяет использовать EXECUTE IMMEDIATE с фиксированной секцией USING.

Реализация могла бы базироваться на DBMS_SQL от начала до конца, но использование DBMS_SQL для выборки строк и значений отдельных полей требует слишком большого объема работы. Гораздо проще было бы использовать традиционную статическую выборку и даже BULK COLLECT. Именно такое решение продемонстрировано в следующем примере:

```
/* Файл в Сети: 11g_to_refcursor.sql */
DECLARE
    TYPE strings_t IS TABLE OF VARCHAR2 (200);

    l_cv          sys_refcursor;
    l_placeholders strings_t    := strings_t ('dept_id');
    l_values      strings_t    := strings_t ('20');
    l_names       strings_t;

    FUNCTION employee_names (
        where_in      IN    VARCHAR2
        , bind_variables_in IN strings_t
        , placeholders_in IN strings_t
    )
    RETURN sys_refcursor
    IS
        l_dyn_cursor NUMBER;
        l_cv          sys_refcursor;
        l_dummy       PLS_INTEGER;
    BEGIN
        /* Разбор курсора для получения фамилий после присоединения
        секции WHERE.
        ВНИМАНИЕ: если вам когда-либо потребуется писать подобный код,
        ОБЯЗАТЕЛЬНО примите меры по снижению риска внедрения SQL.
        Эта тема рассматривается в текущей главе. ПРОЧИТАЙТЕ!
        */
        l_dyn_cursor := DBMS_SQL.open_cursor;
        DBMS_SQL.parse (l_dyn_cursor
            , 'SELECT last_name FROM employees WHERE ' || where_in
            , DBMS_SQL.native
        );

        /*
        Каждая переменная связывается с именованным формальным
        параметром; при переменном количестве формальных параметров
        выполнить этот шаг посредством EXECUTE IMMEDIATE не удастся!
        */
        FOR indx IN 1 .. placeholders_in.COUNT
        LOOP
            DBMS_SQL.bind_variable (l_dyn_cursor
                , placeholders_in (indx)
                , bind_variables_in (indx)
            );
        END LOOP;

        /*
        Выполнение запроса
        */
        l_dummy := DBMS_SQL.EXECUTE (l_dyn_cursor);
        /*
        Преобразование в курсорную переменную, чтобы внешний интерфейс
        или другая программа PL/SQL могли легко получить значения.
        */
        l_cv := DBMS_SQL.to_refcursor (l_dyn_cursor);
        /*
        Курсор не должен закрываться средствами DBMS_SQL; все операции
        с курсором на этой стадии выполняются ТОЛЬКО через
        курсорную переменную.
        */
    END;
```

```

        DBMS_SQL.close_cursor (l_dyn_cursor);
    */
    RETURN l_cv;
END employee_names;
BEGIN
    l_cv := employee_names ('DEPARTMENT_ID = :dept_id', l_values, l_placeholders);

    FETCH l_cv BULK COLLECT INTO l_names;

    CLOSE l_cv;

    FOR indx IN 1 .. l_names.COUNT
    LOOP
        DBMS_OUTPUT.put_line (l_names(indx));
    END LOOP;
END;
/

```

Другой пример ситуации, в которой может пригодиться эта функция: допустим, в программе выполняется динамический SQL, требующий применения DBMS_SQL, но результат должен быть передан клиенту промежуточного уровня (например, приложению Java или .NET). Передать курсор DBMS_SQL невозможно, но ничто не мешает вернуть курсорную переменную.

Функция DBMS_SQL.TO_CURSOR

Функция DBMS_SQL.TO_CURSOR преобразует переменную REF CURSOR (сильно- или слабо-типизированную) в номер курсора SQL, который затем может передаваться подпрограммам DBMS_SQL. Курсорная переменная должна быть открыта до ее передачи функции DBMS_SQL.TO_CURSOR.

После преобразования курсорной переменной в курсор DBMS_SQL вы уже не сможете использовать встроенные операции динамического SQL для обращения к курсору или тем данным, на которых он базируется.

Эта функция будет удобна в том случае, если количество подставляемых в команду SQL переменных известно во время компиляции, но вы не знаете количество элементов, участвующих в выборке (очередной пример динамического SQL категории 4!).

Следующий пример демонстрирует ее практическое применение:

```

/* Файл в Сети: 11g_to_cursorid.sql */
PROCEDURE show_data (
    column_list_in      VARCHAR2
, department_id_in    IN   employees.department_id%TYPE
)
IS
    sql_stmt    CLOB;
    src_cur     SYS_REFCURSOR;
    curid       NUMBER;
    desc_tab    DBMS_SQL.desc_tab;
    colcnt      NUMBER;
    namevar     VARCHAR2 (50);
    numvar      NUMBER;
    datevar     DATE;
    empno       NUMBER           := 100;
BEGIN
    /* Конструирование запроса с внедрением списка выбираемых столбцов
       и одной подставляемой переменной.

```

ВНИМАНИЕ: подобная конкатенация создает угрозу внедрения SQL! Обязательно прочитайте раздел текущей главы, посвященный снижению уязвимости приложения перед подобными атаками.

```

*/
sql_stmt :=
    'SELECT '
    || column_list_in
    || ' FROM employees WHERE department_id = :dept_id';
/* Открытие курсорной переменной с привязкой единственного значения.
   Реализуется НАМНОГО проще, чем с применением DBMS_SQL!
*/
OPEN src_cur FOR sql_stmt USING department_id_in;
/*
Однако курсорная переменная не может использоваться для выборки данных,
потому что количество выбираемых элементов неизвестно во время
компиляции. С другой стороны, проблема идеально решается средствами
DBMS_SQL и процедуры DESCRIBE_COLUMNS, поэтому курсорная переменная
преобразуется в курсор DBMS_SQL, после чего выполняются необходимые
действия.
*/
curid := DBMS_SQL.to_cursor_number (src_cur);
DBMS_SQL.describe_columns (curid, colcnt, desctab);

FOR indx IN 1 .. colcnt
LOOP
    IF desctab (indx).col_type = 2
    THEN
        DBMS_SQL.define_column (curid, indx, numvar);
    ELSIF desctab (indx).col_type = 12
    THEN
        DBMS_SQL.define_column (curid, indx, datevar);
    ELSE
        DBMS_SQL.define_column (curid, indx, namevar, 100);
    END IF;
END LOOP;

WHILE DBMS_SQL.fetch_rows (curid) > 0
LOOP
    FOR indx IN 1 .. colcnt
    LOOP
        DBMS_OUTPUT.put_line (desctab (indx).col_name || ' = ');

        IF (desctab (indx).col_type = 2)
        THEN
            DBMS_SQL.COLUMN_VALUE (curid, indx, namevar);
            DBMS_OUTPUT.put_line (' ' || namevar);
        ELSIF (desctab (indx).col_type = 12)
        THEN
            DBMS_SQL.COLUMN_VALUE (curid, indx, numvar);
            DBMS_OUTPUT.put_line (' ' || datevar);
        ELSE /* Предполагается строка. */
            DBMS_SQL.COLUMN_VALUE (curid, indx, namevar);
            DBMS_OUTPUT.put_line (' ' || namevar);
        ELSEND IF;
    END LOOP;
END LOOP;

DBMS_SQL.close_cursor (curid);
END;
```

Усовершенствованная модель безопасности DBMS_SQL

В 2006 году специалисты в области безопасности обнаружили новый класс уязвимостей, в котором инициирование исключения программой, использующей DBMS_SQL, позволяло атакующему использовать незакрытый курсор для взлома безопасности базы данных.

В Oracle11 для защиты от атак такого рода в пакете DBMS_SQL были введены три дополнительные меры безопасности:

- Генерирование непредсказуемых номеров курсоров.
- Ограничение возможностей использования пакета DBMS_SQL при передаче недействительного номера курсора.
- Отказ в выполнении операции DBMS_SQL, если пользователь, пытающийся работать с курсором, отличен от пользователя, открывшего курсор.

Непредсказуемые номера курсоров

До выхода Oracle Database 11g, вызовы DBMS_SQL.OPEN_CURSOR возвращали последовательно увеличивающиеся числа (обычно из диапазона от 1 до 300). Такая предсказуемость могла позволить атакующему перебрать целые числа и проверить, не являются ли они номерами действительных, открытых курсоров. Обнаруженный курсор мог использоваться для атаки. Сейчас атакующему будет очень трудно найти допустимый курсор посредством перебора. Для примера приведем пять номеров курсоров, возвращаемых OPEN_CURSOR из следующего блока:

```
SQL> BEGIN
2   FOR indx IN 1 .. 5
3   LOOP
4       DBMS_OUTPUT.put_line (DBMS_SQL.open_cursor ());
5   END LOOP;
6 END;
7 /
1693551900
1514010786
1570905132
182110745
1684406543
```

Отказ в доступе к DBMS_SQL при использовании недопустимого номера курсора (ORA-24971)

Чтобы защититься от подбора действительных курсоров атакующим, Oracle теперь немедленно отказывает в доступе к пакету DBMS_SQL при попытке использования недействительного курсора.

Рассмотрим следующий блок:

```
/* Файл в Сети: 11g_access_denied_1.sql */
DECLARE
    l_cursor    NUMBER;
    l_feedback  NUMBER;
    PROCEDURE set_salary
    IS
    BEGIN
        DBMS_OUTPUT.put_line ('Set salary = salary...');
        l_cursor := DBMS_SQL.open_cursor ();
        DBMS_SQL.parse (l_cursor
                        , 'update employees set salary = salary'
                        , DBMS_SQL.native
                        );
        l_feedback := DBMS_SQL.EXECUTE (l_cursor);
        DBMS_OUTPUT.put_line ('  Rows modified = ' || l_feedback);
        DBMS_SQL.close_cursor (l_cursor);
    END set_salary;
BEGIN
    set_salary ();
```

продолжение ➤

```

BEGIN
  l_feedback := DBMS_SQL.EXECUTE (1010101010);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack ());
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace ());
END;

set_salary ();
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack ());
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace ());
END;

```

Здесь выполняется действительная команда UPDATE, которая «обновляет» salary текущим значением для всех строк таблицы employees в локальной процедуре set_salary. Я вызываю эту процедуру, затем пытаюсь выполнить недействительный курсор, после чего снова вызываю set_salary. Результаты выполнения этого блока:

```

Set salary = salary...
Rows modified = 106

```

```

ORA-29471: DBMS_SQL access denied
ORA-06512: at "SYS.DBMS_SQL", line 1501
ORA-06512: at line 22
Set salary = salary...
ORA-29471: DBMS_SQL access denied
ORA-06512: at "SYS.DBMS_SQL", line 980
ORA-06512: at line 9
ORA-06512: at line 30

```

Процедура set_salary сработала в первый раз, но после попытки выполнения недействительного курсора при повторном запуске set_salary было инициировано исключение ORA-29471. Собственно, любая попытка вызова программы DBMS_SQL приведет к выдаче этой ошибки.

Восстановить доступ к DBMS_SQL можно только одним способом: выходом и повторным подключением. Да, жестко! Но это вполне разумно с учетом потенциально опасной ситуации, приведшей к ошибке.

База данных также блокирует доступ к DBMS_SQL, если программа, в которой был открыт курсор, инициировала исключение (которое может быть не связано с динамическим SQL). Если ошибка «поглощается» программой (то есть не инициируется заново), определить ее причину будет нелегко.

Запрет операций DBMS_SQL при изменении действующего пользователя (ORA-24970)

Oracle Database 11g предоставляет новую перегруженную версию функции OPEN_CURSOR с аргументом:

```
DBMS_SQL.OPEN_CURSOR (уровень_безопасности IN INTEGER) RETURN INTEGER;
```

Эта функция позволяет задать уровень безопасности, который будет обеспечиваться Oracle для открытого курсора при выполнении операций с этим курсором. Ниже перечислены уровни безопасности, поддерживаемые базой данных в настоящее время:

- 0 — отключение проверок безопасности для операций DBMS_SQL с курсором. Это означает, что вы можете выполнить выборку из курсора, а затем выполнить подстановки и заново выполнить курсор с другим фактическим идентификатором пользователя

или ролью (отличными от действовавших на момент первого разбора). Этот уровень безопасности не действует по умолчанию.

- 1 — фактические идентификаторы пользователя и роли вызывающей стороны DBMS_SQL для подстановки и выполнения операций с курсором должны совпадать с тем, которые действовали для вызывающей стороны при последней операции разбора с этим курсором.
- 2 — фактические идентификаторы пользователя и роли вызывающей стороны DBMS_SQL для всех операций подстановки, выполнения, определения, описания и выборки с курсором должны совпадать с тем, которые действовали для вызывающей стороны при последней операции разбора с этим курсором.

Пример возможных ошибок, обусловленных новыми проверками безопасности Oracle:

1. Создайте процедуру `user_cursor` в схеме `HR`, как показано ниже. Обратите внимание: эта программа выполняется с правами создателя; это означает, что при вызове этой программы другой схемой текущим или фактическим пользователем становится `HR`. Откройте курсор и разберите запрос для `ALL_SOURCE` с этим курсором. Затем верните номер курсора DBMS_SQL в аргументе `OUT`:

```
/* Файл в Сети: 11g_effective_user_id.sql */
PROCEDURE user_cursor (
    security_level_in IN PLS_INTEGER
, cursor_out IN OUT NUMBER
)
AUTHID DEFINER
IS
BEGIN
    cursor_out := DBMS_SQL.open_cursor (security_level_in);
    DBMS_SQL.parse (cursor_out
, 'select count(*) from all_source'
, DBMS_SQL.native
);
END;
```

2. Предоставьте разрешение на запуск программы пользователю `SCOTT`:

```
GRANT EXECUTE ON use_cursor TO scott
```

3. Подключитесь с правами `SCOTT`. Запустите программу `use_cursor` из схемы `HR` с безопасностью уровня 2 и получите динамический курсор SQL. Наконец, попытайтесь выполнить этот курсор из схемы `SCOTT`:

```
SQL> DECLARE
2   l_cursor NUMBER;
3   l_feedback number;
4   BEGIN
5   hr.use_cursor (2, l_cursor);
6   l_feedback := DBMS_SQL.execute_and_fetch (l_cursor);
7   END;
8   /
```

```
DECLARE
```

```
*
```

```
ERROR at line 1:
```

```
ORA-29470: Effective userid or roles are not the same as when cursor was parsed
```

```
ORA-06512: at "SYS.DBMS_SQL", line 1513
```

```
ORA-06512: at line 6
```

Oracle выдает ошибку ORA-29470, потому что курсор был открыт и разобран в схеме `HR` (из-за присутствия `AUTHID DEFINER`), но выполнялся в схеме `SCOTT`.

Создание приложений PL/SQL

В этой части книги мы сводим воедино все, о чем говорилось ранее. К этому времени вы уже умеете объявлять переменные и работать с ними, знаете, как обрабатывать ошибки и создавать циклы. Словом, вы полностью готовы к тому, чтобы заняться построением приложений. Именно этой теме и посвящена данная часть книги. Сначала мы научимся создавать процедуры, функции, пакеты и триггеры, а затем рассмотрим возможности управления приложением в целом.

17

Процедуры, функции и параметры

В предыдущих частях книги подробно рассматривались основные компоненты и конструкции языка PL/SQL: курсоры, исключения, циклы, переменные и т. д. Однако для создания хорошо структурированного, легкого для понимания и сопровождения приложения уметь работать с этими компонентами недостаточно — нужно правильно компоновать все части кода.

Лишь немногие задачи имеют тривиальное решение, которое можно сразу понять и немедленно реализовать на бумаге (или на клавиатуре). Большинство проектируемых нами систем объемны и сложны, в них входит много взаимодействующих и даже конфликтующих компонентов. Кроме того, новые приложения значительно мощнее своих предшественников, что сопровождается еще большим усложнением их внутренней реализации.

Сейчас одной из самых важных задач в профессии программиста является упрощение рабочей среды. Столкнувшись с исключительно сложной задачей, наш разум приходит в смятение. С чего начать? Как пробраться через эти джунгли требований и необходимой функциональности?

Человеческий разум не сравнится с большим параллельным компьютером. Даже самые выдающиеся умы едва ли смогут удерживать в памяти пять–семь задач одновременно. Большие проекты приходится разбивать на меньшие компоненты, а те, в свою очередь, делятся на отдельные программы. И лишь после создания этих программ мы сможем сконструировать из них полнофункциональные приложения.

Если воспользоваться *нисходящим* методом проектирования (то есть принципом *пошаговой проработки*, подробно описанным в разделе «Локальные модули» этой главы) или другой методологией, разбиение кода на процедуры, функции и объектные типы позволит создать качественное и легкое в сопровождении приложение.

Модульный код

Модуляризацией называется разбиение больших блоков кода на меньшие блоки (модули), которые можно вызывать из других модулей. Этот процесс сходен с нормализацией данных; он обладает теми же преимуществами и рядом дополнительных достоинств. В частности, модуляризация позволяет улучшить ряд характеристик кода:

- **Пригодность для повторного использования.** Разбивая большие программы на отдельные взаимодействующие компоненты, вы увидите, что многие модули ис-

пользуются более чем одной программой текущего приложения. При правильной организации такие программы даже могут пригодиться в других приложениях!

- **Управляемость.** Что удобнее отлаживать: одну программу в 10 000 строк или пять отдельных программ по 2000 строк, которые вызывают друг друга при необходимости? Наш разум лучше приспособлен к решению небольших задач. К тому же при использовании модульного подхода код можно протестировать и отладить на уровне отдельных программ (так называемое модульное тестирование), то есть до того, как отдельные модули будут скомбинированы для проведения более сложных интеграционных тестов.
- **Удобство чтения.** Имена модулей отражают их назначение. Чем больше кода будет перемещено в программный интерфейс или скрыто за ним, тем легче будет понять, что делает программа. Модуляризация дает возможность сосредоточиться на задаче в целом, а не на отдельных фрагментах кода. Возможно, вам даже удастся реализовать одну из самых сложных целей: создать самодокументируемый код.
- **Надежность.** Код, разбитый на модули, содержит меньше ошибок, а обнаруженные ошибки легче исправлять, так как они локализованы на уровне модуля. Кроме того, такой код легче сопровождать другим программистам, он имеет меньший объем и лучше читается.

Поскольку вы уже изучили все основные конструкции PL/SQL (циклы, условные переходы и т. д.), то в принципе можете браться за написание программ. Однако для написания качественного приложения необходимо уметь создавать и комбинировать модули.

В PL/SQL существует несколько конструкций для разбиения кода на модули:

- **Процедура.** Программа, которая осуществляет одно или несколько действий и вызывается как исполняемый оператор PL/SQL. Передавать данные процедуре и получать их можно с помощью списка параметров.
- **Функция.** Программа, которая возвращает одно значение и используется как выражение PL/SQL. Для передачи информации функции используется ее список параметров. Параметры также могут использоваться для возврата информации из функции, но обычно это считается проявлением плохого стиля программирования.
- **Триггер базы данных.** Набор команд, который вызывается при выполнении некоторого события (подключение к базе данных, модификация строки таблицы или DDL-операция).
- **Пакет.** Именованный набор процедур, функций, типов и переменных. Пакет не является модулем (скорее, это мета-модуль), но он тесно связан с реализацией модульного подхода.
- **Объектный тип или экземпляр объектного типа.** Эмуляция объектно-ориентированного класса в Oracle. Объектный тип инкапсулирует состояние и поведение данных, комбинируя их (как реляционная таблица) с правилами (процедурами и функциями, которые манипулируют этими данными).

Пакеты рассматриваются в главе 18, триггеры баз данных — в главе 19, а объектные типы — в главе 26. В этой главе мы займемся процедурами и функциями, а также принципами создания их списков параметров, являющихся неотъемлемой частью хорошо спроектированных модулей.

Термин «модуль» в этом контексте может обозначать как функцию, так и процедуру. Как и во многих других языках программирования, в PL/SQL модули могут вызывать другие именованные модули. Для передачи и получения информации из модулей используются параметры. Кроме того, модульная структура PL/SQL хорошо сочетается с реализацией обработчиков исключений и предоставляет программисту все возможности по части обработки ошибок (см. главу 6).

В этой главе рассказано, как объявляются процедуры и функции и как устанавливаются списки параметров для них. Кроме того, мы исследуем такие «экзотические» аспекты разработки программ, как локальные модули, перегрузка, опережающие ссылки и детерминированные функции.

Процедуры

Процедура представляет собой модуль, выполняющий одно или несколько действий. Поскольку вызов процедуры в PL/SQL является отдельным исполняемым оператором, блок кода PL/SQL может состоять только из вызова процедуры. Процедуры относятся к числу ключевых компонентов модульного кода, обеспечивающих оптимизацию и повторное использование программной логики.

Общий формат процедуры PL/SQL выглядит так:

```
PROCEDURE [схема.]имя[( параметр[, параметр...] ) ]
  [AUTHID DEFINER | CURRENT_USER]
  [ACCESSIBLE BY (список)]
IS
  [объявления]
BEGIN
  исполняемые команды
[ EXCEPTION
  обработчики исключений]
END [имя];
```

Основные элементы этой структуры:

- *схема* — имя схемы, которой будет принадлежать процедура (необязательный аргумент). По умолчанию применяется имя схемы текущего пользователя. Если значение схемы отлично от имени схемы текущего пользователя, то этот пользователь должен обладать привилегиями для создания процедуры в другой схеме.
- *имя* — имя процедуры.
- *параметр* — необязательный список параметров, которые применяются для передачи данных в процедуру и возврата информации из процедуры в вызывающую программу.
- *AUTHID* — определяет, с какими разрешениями будет вызываться процедура: создателя (владельца) или текущего пользователя. В первом случае процедура выполняется с правами создателя, во втором — с правами вызывающего. Эти две модели подробно описаны в главе 24.
- *объявления* — объявления локальных идентификаторов этой процедуры. Если объявления отсутствуют, между ключевыми словами *IS* и *BEGIN* не будет никаких выражений.
- *ACCESSIBLE BY* (Oracle Database 12c) — ограничивает доступ к процедуре программными модулями, перечисленными в круглых скобках. Данная возможность рассматривается в главе 24.
- *исполняемые команды* — команды, выполняемые процедурой при вызове. Между ключевыми словами *BEGIN* и *END* или *EXCEPTION* должна находиться по крайней мере одна исполняемая команда.
- *обработчики исключений* — необязательные обработчики исключений для процедуры. Если процедура не обрабатывает никаких исключений, слово *EXCEPTION* можно опустить и завершить исполняемый раздел ключевым словом *END*.

На рис. 17.1 показан код процедуры `apply_discount`, который содержит все четыре раздела, характерных для именованных блоков PL/SQL.

```

PROCEDURE apply_discount
(company_id_in IN company.company_id%TYPE, discount_in IN NUMBER)
IS
min_discount CONSTANT NUMBER:=.05;
max_discount CONSTANT NUMBER:=.25;
invalid_discount EXCEPTION;
BEGIN
IF discount_in BETWEEN min_discount AND max_discount
THEN
UPDATE item
SET item_amount:=item_amount*(1-discount_in);
WHERE EXISTS (SELECT 'x' FROM order
WHERE order.order_id=item.order_id
AND order.company_id=company_id_in);
IF SQL%ROWCOUNT = 0 THEN RAISE NO_DATA_FOUND; END IF;
ELSE
RAISE invalid_discount;
END IF;
EXCEPTION
WHEN invalid_discount
THEN
DBMS_OUTPUT.PUT_LINE('The specified discount is invalid.');
```

Заголовок

Объявление

Исполняемый раздел

Раздел исключений

```

WHEN NO_DATA_FOUND
THEN
DBMS_OUTPUT.PUT_LINE('No orders in the system for company:'||
TO_CHAR(company_id_in));
END apply_discount;
```

Рис. 17.1. Код процедуры

Вызов процедуры

Процедура вызывается как исполняемая команда PL/SQL. Другими словами, ее вызов должен заканчиваться точкой с запятой (;) и может предшествовать другим командам SQL либо PL/SQL (если таковые имеются) в исполняемом разделе блока PL/SQL или следовать за ними:

```

BEGIN
  apply_discount( new_company_id, 0.15 );
END;
```

Если процедура не имеет параметров, она может вызываться с пустыми круглыми скобками или без них:

```

display_store_summary;
display_store_summary();
```

Заголовок процедуры

Часть определения процедуры, предшествующая ключевому слову IS, называется *заголовком процедуры*, или *сигатурой*. Заголовок предоставляет программисту всю информацию, необходимую для вызова процедуры:

- Имя процедуры.
- Условие AUTHID (если имеется).

- Список параметров (если имеется).
- Список ACCESSIBLE BY (если имеется — новая возможность Oracle Database 12c).

В идеале программист при виде заголовка процедуры должен понять, что делает эта процедура и как она вызывается.

Заголовок процедуры `apply_discount` из предыдущего раздела выглядит так:

```
PROCEDURE apply_discount (  
    company_id_in IN company.company_id%TYPE  
    , discount_in IN NUMBER  
)
```

Он состоит из типа модуля, имени и списка из двух параметров.

Тело процедуры

В теле процедуры содержится код, необходимый для реализации этой процедуры; тело состоит из объявления, исполняемого раздела и раздела исключений этой процедуры. Все, что следует за ключевым словом `IS`, образует тело процедуры. Разделы исключений и объявлений не являются обязательными. Если обработчики исключений отсутствуют, опустите ключевое слово `EXCEPTION` и завершите процедуру командой `END`.

Метка END

Вы можете указать имя процедуры за завершающим ключевым словом `END`:

```
PROCEDURE display_stores (region_in IN VARCHAR2) IS  
BEGIN  
    ...  
END display_stores;
```

Имя служит меткой, явно связывающей конец программы с ее началом. Привыкните к использованию метки `END`. Она особенно полезна для процедур, занимающих несколько страниц или входящих в серию процедур и функций в теле пакета.

Команда RETURN

Ключевое слово `RETURN` обычно ассоциируется с функциями, поскольку они должны возвращать значения. Однако PL/SQL позволяет использовать команду `RETURN` в процедурах. Версия этой команды для процедур не принимает выражений и не может возвращать значения в вызывающий программный модуль — она просто прекращает выполнение процедуры и возвращает управление вызывающему коду.

Использовать эту разновидность `RETURN` не рекомендуется, поскольку в этом случае в процедуре появляются две и более точки выхода, а это усложняет логику выполнения. Избегайте использования `RETURN` и `GOTO` для обхода нормальной управляющей структуры в программах.

Функции

Функция представляет собой модуль, который возвращает значение командой `RETURN` (вместо аргументов `OUT` или `IN OUT`). В отличие от вызова процедуры, который представляет собой отдельный оператор, вызов функции всегда является частью исполняемого оператора, то есть включается в выражение или служит в качестве значения по умолчанию, присваиваемого переменной при объявлении.

Возвращаемое функцией значение принадлежит к определенному типу данных. Функция может использоваться вместо выражения, которое имеет тот же тип данных, что и возвращаемое ею значение.

Функции играют важную роль в реализации модульного подхода. К примеру, реализацию отдельного бизнес-правила или формулы в приложении рекомендуется оформить в виде функции. Вместо того чтобы писать один и тот же запрос снова и снова («Получить имя работника по идентификатору», «Получить последнюю строку заказа из таблицы `order` для заданного идентификатора компании» и т. д.), поместите его в функцию и вызовите эту функцию в нужных местах. Такой код создает меньше проблем с отладкой, оптимизацией и сопровождением.



Некоторые программисты предпочитают вместо функций использовать процедуры, возвращающие информацию через список параметров. Если вы принадлежите к их числу, проследите за тем, чтобы ваши бизнес-правила, формулы и однострочные запросы были скрыты в процедурах!

В приложениях, которые не определяют и не используют функции, со временем обычно возникают трудности с сопровождением и расширением.

Структура функции

Функция (рис. 17.2) имеет почти такую же структуру, как и процедура, не считая того, что ключевое слово `RETURN` в ней играет совершенно другую роль:

```
FUNCTION [схема.]имя[( параметр[, параметр...]) ]
    RETURN возвращаемый тип
    [AUTHID DEFINER | CURRENT_USER]
    [DETERMINISTIC]
    [PARALLEL_ENABLE ...]
    [PIPELINED]
    [RESULT_CACHE ...]
    [ACCESSIBLE BY (program_unit_list)]
    [AGGREGATE ...]
    [EXTERNAL ...]
IS
    [объявления]
BEGIN
    исполняемые команды
[EXCEPTION
    обработчики исключений]
END [имя];
```

Основные элементы этой структуры:

- *схема* — имя схемы, которой будет принадлежать функция (необязательный аргумент). По умолчанию применяется имя схемы текущего пользователя. Если значение схемы отлично от имени схемы текущего пользователя, то этот пользователь должен обладать привилегиями для создания функции в другой схеме.
- *имя* — имя функции.
- *параметр* — необязательный список параметров, которые применяются для передачи данных в функцию и возврата информации из нее в вызывающую программу.
- *возвращаемый тип* — задает тип значения, возвращаемого функцией. Возвращаемый тип должен быть указан в заголовке функции; он более подробно рассматривается в следующем разделе.
- `AUTHID` — определяет, с какими разрешениями будет вызываться функция: создателя (владельца) или текущего пользователя. В первом случае (используется по умолчанию) применяется модель прав создателя, во втором — модель прав вызывающего.
- `DETERMINISTIC` — определяет функцию как *детерминированную*, то есть возвращаемое значение полностью определяется значениями ее аргументов. Если включить эту секцию, ядро SQL сможет оптимизировать выполнение функции при

ее вызове в запросах. За дополнительной информацией обращайтесь к разделу «Детерминированные функции» этой главы.

- **PARALLEL_ENABLE** — используется для оптимизации и позволяет функции выполняться параллельно в случае, когда она вызывается из команды **SELECT**.
- **PIPELINED** — указывает, что результат табличной функции должен возвращаться в итеративном режиме с помощью команды **PIPE ROW**.
- **RESULT_CACHE** — указывает, что входные значения и результат вызова функции должен быть сохранен в кэше результатов. Эта возможность, появившаяся в Oracle11g, более подробно рассматривается в главе 21.
- **ACCESSIBLE BY (Oracle Database 12c)** — ограничивает доступ к функции программными модулями, перечисленными в круглых скобках. Данная возможность рассматривается в главе 24.
- **AGGREGATE** — используется при определении агрегатных функций.
- **EXTERNAL** — определяет функцию с «внешней реализацией» — то есть написанную на языке C.
- *объявления* — объявления локальных идентификаторов этой функции. Если объявления отсутствуют, между ключевыми словами **IS** и **BEGIN** не будет никаких выражений.
- *исполняемые команды* — команды, выполняемые функцией при вызове. Между ключевыми словами **BEGIN** и **END** или **EXCEPTION** должна находиться по крайней мере одна исполняемая команда.

```

FUNCTION tot_sales
(company_id_in IN company.company_id%TYPE,
status_in IN order.status_code%TYPE:=NULL)
RETURN NUMBER
IS
/*Internal upper-cased version of status code */
status_int order.status_code%TYPE:=UPPER(status_in);

/*Parameterized cursor returns total discounted sales.*/
CURSOR sales_cur (status_in IN status_code%TYPE) IS
SELECT SUM (amount*discount)
FROM item
WHERE EXISTS (SELECT 'X' FROM order
WHERE order.order_id=item.order_id
AND company_id=company_id_in
AND status_code LIKE status_in);

/*Return value for function*/
return_value NUMBER;
BEGIN
OPEN sales_cur (status_int);
FETCH sales_cur INTO return_value;
IF sales_cur%NOTFOUND
THEN
CLOSE sales_cur;
RETURN NULL;
ELSE
CLOSE sales_cur;
RETURN return_value;
END IF;
END tot_sales;

```

Заголовок

Объявление

Исполняемый раздел

Рис. 17.2. Код функции

- *обработчики исключений* — необязательные обработчики исключений для функции. Если процедура не обрабатывает никаких исключений, слово `EXCEPTION` можно опустить и завершить исполняемый раздел ключевым словом `END`.

На рис. 17.2 изображено строение функции PL/SQL и ее различных разделов. Обратите внимание: функция `total_sales` не имеет раздела исключений.

Возвращаемый тип

Функция PL/SQL может возвращать данные практически любого типа, поддерживаемого PL/SQL, — от скаляров (единичных значений вроде даты или строки) до сложных структур: коллекций, объектных типов, курсорных переменных и т. д.

Несколько примеров использования `RETURN`:

- Возвращение строки:

```
FUNCTION favorite_nickname (
    name_in IN VARCHAR2) RETURN VARCHAR2
IS
BEGIN
    ...
END;
```

- Возвращение числа функцией-членом объектного типа:

```
TYPE pet_t IS OBJECT (
    tag_no          INTEGER,
    NAME            VARCHAR2 (60),
    breed           VARCHAR2(100),
    dob DATE,
    MEMBER FUNCTION age RETURN NUMBER
)
```

- Возвращение записи, имеющей ту же структуру, что и у таблицы `books`:

```
PACKAGE book_info
IS
    FUNCTION onerow (isbn_in IN books.isbn%TYPE)
        RETURN books%ROWTYPE;
    ...
```

Возвращение курсорной переменной с заданным типом `REF CURSOR` (базирующемся на типе записи):

```
PACKAGE book_info
IS
    TYPE overdue_rt IS RECORD (
        isbn books.isbn%TYPE,
        days_overdue PLS_INTEGER);

    TYPE overdue_rct IS REF CURSOR RETURN overdue_rt;

    FUNCTION overdue_info (username_in IN lib_users.username%TYPE)
        RETURN overdue_rct;
    ...
```

Метка END

Вы можете указать имя функции за завершающим ключевым словом `END`:

```
FUNCTION total_sales (company_in IN INTEGER) RETURN NUMBER
IS
BEGIN
    ...
END total_sales;
```

Имя служит меткой, явно связывающей конец программы с ее началом. Привыкните к использованию метки END. Она особенно полезна для функций, занимающих несколько страниц или входящих в серию процедур и функций в теле пакета.

Вызов функции

Функция может вызываться из любой части исполняемой команды PL/SQL, где допускается использование выражения. Следующие примеры демонстрируют вызовы функций, определения которых приводились в предыдущем разделе.

- Присваивание переменной значения по умолчанию вызовом функции:

```
DECLARE
    v_nickname VARCHAR2(100) :=
        favorite_nickname ('Steven');
```

- Использование функции-члена для объектного типа в условии:

```
DECLARE
    my_parrot pet_t :=
        pet_t (1001, 'Mercury', 'African Grey',
            TO_DATE ('09/23/1996', 'MM/DD/YYYY'));
BEGIN
    IF my_parrot.age () < INTERVAL '50' YEAR
    THEN
        DBMS_OUTPUT.PUT_LINE ('Still a youngster!');
    END IF;
```

- Вставка в запись строки с информацией о книге:

```
DECLARE
    my_first_book books%ROWTYPE;
BEGIN
    my_first_book := book_info.onerow ('1-56592-335-9');
    ...
```

- Вызов пользовательской функции PL/SQL из запроса:

```
DECLARE
    l_name employees.last_name%TYPE;
BEGIN
    SELECT last_name INTO l_name
    FROM employees
    WHERE employee_id = hr_info_pkg.employee_of_the_month ('FEBRUARY');
    ...
```

Вызов написанной вами функции из команды CREATE VIEW с использованием выражения CURSOR для передачи результирующего набора в аргументе функции:

```
VIEW young_managers
AS
    SELECT managers.employee_id AS manager_employee_id
    FROM employees managers
    WHERE most_reports_before_manager
        (
            CURSOR ( SELECT reports.hire_date
                    FROM employees reports
                    WHERE reports.manager_id = managers.employee_id
                ),
            managers.hire_date
        ) = 1;
```

В PL/SQL, в отличие от некоторых других языков программирования, невозможно просто проигнорировать возвращаемое значение функции, даже если оно не представляет интереса для вас. Например, для следующего вызова функции:

```
BEGIN
    favorite_nickname('Steven');
END;
```

будет выдана ошибка *PLS-00221: 'FAVORITE_NICKNAME' is not a procedure or is undefined.* Функцию нельзя использовать так, как если бы она была процедурой.

Функции без параметров

Если функция не имеет параметров, ее вызов может записываться с круглыми скобками или без них. Следующий код демонстрирует эту возможность на примере вызова метода `age` объектного типа `pet_t`:

```
IF my_parrot.age < INTERVAL '50' YEAR -- 9i INTERVAL type
IF my_parrot.age() < INTERVAL '50' YEAR
```

Заголовок функции

Часть определения функции, предшествующая ключевому слову **IS**, называется *заголовком функции*, или *сигатурой*. Заголовок предоставляет программисту всю информацию, необходимую для вызова функции:

- Имя функции.
- Модификаторы определения и поведения функции (детерминированность, возможность параллельного выполнения и т. д.).
- Список параметров (если имеется).
- Тип возвращаемого значения.

В идеале программист при виде заголовка функции должен понять, что делает эта функция и как она вызывается.

Заголовок упоминавшейся ранее функции `total_sales` выглядит так:

```
FUNCTION total_sales
    (company_id_in IN company.company_id%TYPE,
     status_in IN order.status_code%TYPE := NULL)
RETURN NUMBER
```

Он состоит из типа модуля, имени и списка из двух параметров и возвращаемого типа `NUMBER`. Это означает, что любое выражение или команда PL/SQL, в которых задействовано числовое значение, может вызвать `total_sales` для получения этого значения. Пример:

```
DECLARE
    v_sales NUMBER;
BEGIN
    v_sales := total_sales (1505, 'ACTIVE');
    ...
END;
```

Тело функции

В *теле функции* содержится код, необходимый для реализации этой функции; тело состоит из объявления, исполняемого раздела и раздела исключений этой функции. Все, что следует за ключевым словом **IS**, образует тело функции.

Как и в случае с процедурами, разделы исключений и объявлений не являются обязательными. Если обработчики исключений отсутствуют, опустите ключевое слово **EXCEPTION** и завершите функцию командой **END**. Если объявления отсутствуют, команда **BEGIN** просто следует непосредственно за ключевым словом **IS**.

Исполняемый раздел функции должен содержать команду `RETURN`. Функция откомпилируется и без него, но если выполнение функции завершится без выполнения команды `RETURN`, Oracle выдаст ошибку: *ORA-06503: PL/SQL: Function returned without value.*



Эта ошибка не выдается, если функция передает наружу свое необработанное исключение.

Команда RETURN

В исполняемом разделе функции должна находиться по меньшей мере одна команда `RETURN`. Команд может быть и несколько, но в одном вызове функции должна выполняться только одна из них. После обработки команды `RETURN` выполнение функции прекращается, и управление передается вызывающему блоку PL/SQL.

Если ключевое слово `RETURN` в заголовке определяет тип данных возвращаемого значения, то команда `RETURN` в исполняемом разделе задает само это значение. При этом тип данных, указанный в заголовке, должен быть совместим с типом данных выражения, возвращаемого командой `RETURN`.

Любое допустимое выражение

Команда `RETURN` может возвращать любое выражение, совместимое с типом, обозначенным в секции `RETURN`. Это выражение может включать вызовы других функций, сложные вычисления и даже преобразования данных. Все следующие примеры использования `RETURN` допустимы:

```
RETURN 'buy me lunch';
RETURN POWER (max_salary, 5);
RETURN (100 - pct_of_total_salary (employee_id));
RETURN TO_DATE ('01' || earliest_month || initial_year, 'DDMMYY');
```

Вы также можете возвращать сложные типы данных — экземпляры объектных типов, коллекции и записи.

Выражение в команде `RETURN` вычисляется в момент выполнения `RETURN`. При возврате управления в вызывающий блок также передается результат вычисленного выражения.

Множественные команды RETURN

В функции `total_sales` на рис. 17.2 я использую две разные команды `RETURN` для обработки разных ситуаций в функции: если из курсора не удалось получить информацию, возвращается `NULL` (не ноль). Если же от курсора было получено значение, оно возвращается вызывающей программе. В обоих случаях команда `RETURN` возвращает значение: в одном случае `NULL`, в другом — переменную `return_value`.

Конечно, наличие нескольких команд `RETURN` в исполняемом разделе функции разрешено, однако лучше ограничиться одной командой `RETURN`, размещаемой в последней строке исполняемого раздела. Причины объясняются в следующем разделе.

RETURN как последняя исполняемая команда

В общем случае команду `RETURN` желательно делать последней исполняемой командой; это лучший способ гарантировать, что функция всегда возвращает значение. Объявите переменную с именем `return_value` (которое четко указывает, что в переменной будет

храниться возвращаемое значение функции), напишите весь код вычисления этого значения, а затем в самом конце функции верните значение `return_value` командой `RETURN`:

```
FUNCTION do_it_all (parameter_list) RETURN NUMBER IS
    return_value NUMBER;
BEGIN
    ... множество исполняемых команд ...
    RETURN return_value;
END;
```

Переработанная версия логики на рис. 17.2, в которой решена проблема множественных команд `RETURN`, выглядит так:

```
OPEN sales_cur;
IF sales_cur%NOTFOUND
THEN
    return_value:= NULL;
END IF;
CLOSE sales_cur;
RETURN return_value;
```

Остерегайтесь исключений! Помните, что инициированное исключение может «перепрыгнуть» через последнюю команду прямо в обработчик. Если обработчик исключения не содержит команды `RETURN`, то будет выдана ошибка *ORA-06503: Function returned without value* независимо от того, как было обработано исходное исключение.

Параметры

Для передачи информации между модулем и вызывающим блоком PL/SQL используются параметры.

Параметры модуля, являющиеся частью его заголовка (или сигнатуры), являются не менее важными компонентами модуля, чем находящиеся в нем исполняемые команды. Заголовок программы иногда называется *контрактом* между автором и пользователями. Конечно, автор должен позаботиться о том, чтобы модуль выполнял свою задачу. Но ведь модули создаются именно для того, чтобы их можно было вызывать повторно — в идеале более чем из одного модуля. Если список параметров плохо составлен, то другим программистам будет сложно применять такой модуль. В таком случае уже будет неважно, насколько хорошо он написан.

Многие разработчики не уделяют должного внимания наборам параметров своих модулей. Следующие рекомендации помогут вам сделать правильный выбор:

- **Количество параметров.** Если процедура или функция имеет слишком мало параметров, это снижает ее универсальность; с другой стороны, слишком большое количество параметров усложняет ее повторное использование. Конечно, количество параметров в основном определяется требованиями программы, но существуют разные варианты их определения (например, несколько параметров можно объединить в одну запись).
- **Типы параметров.** При выборе типа необходимо учитывать, для каких целей будут использоваться параметры: только для чтения, только для записи, для чтения и записи.
- **Имена параметров.** Параметрам следует присваивать простые имена, отражающие их назначение в модуле.
- **Значения по умолчанию.** Когда параметру следует задать значение по умолчанию, а когда нужно заставить программиста ввести определенное значение?

PL/SQL предоставляет много средств эффективного планирования. В этом разделе представлены все элементы определения параметров.

Определение параметров

Формальные параметры определяются в списке параметров программы. Синтаксис определения параметров очень близок к синтаксису объявления переменных в разделе объявлений блока PL/SQL. Впрочем, существуют два важных различия: во-первых, для параметров определяется *режим передачи*, которого нет у объявлений переменных; во-вторых, объявление параметра должно быть неограниченным.

Ограниченное объявление устанавливает некоторые ограничения для значений, которые могут присваиваться переменной, объявленной с этим типом. Например, следующее объявление переменной `company_name` ограничивает переменную 60 символами:

```
DECLARE
    company_name VARCHAR2(60);
```

При объявлении параметра ограничивающая часть опускается:

```
PROCEDURE display_company (company_name IN VARCHAR2) IS ...
```

Формальные и фактические параметры

Очень важно понимать различия между формальными и фактическими параметрами. *Формальные параметры* представляют собой имена, объявленные в списке параметров заголовка модуля, тогда как *фактические параметры* — это значения и выражения, которые помещаются в список параметров при вызове модуля.

Различия между формальными и фактическими параметрами нам также поможет понять функция `total_sales`. Ее заголовок выглядит так:

```
FUNCTION total_sales
    (company_id_in IN company.company_id%TYPE,
     status_in IN order.status_code%TYPE DEFAULT NULL)
RETURN std_types.dollar_amount;
```

Формальные параметры `total_sales`:

`company_id_in` — первичный ключ (идентификатор компании).

`status_in` — статус заказов, включаемых в вычисление.

Эти формальные параметры не существуют за пределами модуля. Их можно рассматривать как своего рода «заместителей» реальных значений, передаваемых модулю при его использовании в программе (то есть фактических параметров).

При вызове `total_sales` необходимо предоставить два аргумента, которыми могут быть переменные, константы или литералы (для параметров в режимах `OUT` и `IN OUT` это должны быть переменные). В следующем примере переменная `company_id` содержит первичный ключ, указывающий на запись компании. В первых трех вызовах `total_sales` функции передаются жестко запрограммированные значения статуса заказов, а в последнем вызове статус не указан; в этом случае функция присваивает параметру `status_in` значение по умолчанию, указанное в заголовке:

```
new_sales      := total_sales (company_id, 'N');
paid_sales     := total_sales (company_id, 'P');
shipped_sales  := total_sales (company_id, 'S');
all_sales      := total_sales (company_id);
```

При вызове `total_sales` вычисляются значения всех фактических параметров. Результаты вычислений присваиваются соответствующим формальным параметрам внутри функции (обратите внимание: это относится только к параметрам `IN` и `IN OUT`; параметры режима `OUT` не копируются).

Формальные параметры и соответствующие им фактические параметры (указанные при вызове) должны относиться к одинаковым или совместимым типам данных. Во

многих ситуациях PL/SQL выполняет преобразования данных автоматически, однако лучше по возможности обходиться без неявных преобразований. Используйте такие функции, как `TO_CHAR` (см. главу 7) и `TO_DATE` (см. главу 10), чтобы точно знать, какие данные получает модуль.

Режимы передачи параметров

При определении параметра также указывается допустимый способ его применения. Он задается с помощью одного из трех указанных в таблице режимов.

| Режим | Предназначение | Использование параметров |
|--------|---------------------|---|
| IN | Только для чтения | Значение параметра может применяться, но не может быть изменено в модуле. Если режим параметра не задан, используется режим IN |
| OUT | Только для записи | В модуле можно присвоить значение параметру, но нельзя использовать его. Впрочем, это «официальное» определение — на самом деле Oracle позволяет читать значение параметра OUT в подпрограмме |
| IN OUT | Для чтения и записи | В модуле можно использовать и изменять значение параметра |

Режим параметра указывается непосредственно после его имени, но перед типом данных и необязательным значением по умолчанию. В следующем заголовке процедуры задействованы все три режима передачи параметров:

```
PROCEDURE predict_activity
  (last_date_in IN DATE,
   task_desc_inout IN OUT VARCHAR2,
   next_date_out OUT DATE)
```

Процедура `predict_activity` принимает дату последнего действия (`last_date`) и описание этого действия (`task_desc_inout`). Возвращает она два значения: описание действия (возможно, модифицированное) и дату следующего действия (`next_date_out`). Поскольку параметр `task_desc_inout` передается в режиме IN OUT, программа может читать и изменять его значение.

Режим IN

Параметр IN позволяет передавать значения модулю, но не может использоваться для передачи информации из модуля вызывающему блоку PL/SQL. Другими словами, в контексте вызываемой программы параметры IN работают как константы. Значение формального параметра IN, как и значение константы, не может быть изменено внутри программы. Параметру IN нельзя присвоить новое значение или изменить его иным образом — компилятор выдаст сообщение об ошибке.

Режим IN используется по умолчанию; если режим параметра не задан, параметр автоматически считается определенным в режиме IN. Тем не менее я рекомендую всегда указывать режим параметра, чтобы предполагаемое использование было явно указано в коде.

В заголовке программы параметрам IN могут присваиваться значения по умолчанию (см. раздел «Значения по умолчанию»).

Фактическим значением параметра IN может быть переменная, именованная константа, литерал или сложное выражение. Все следующие вызовы `display_title` допустимы:

```
/* Файл в Сети: display_title.sql */
DECLARE
  happy_title CONSTANT VARCHAR2(30) := 'HAPPY BIRTHDAY';
  changing_title VARCHAR2(30) := 'Happy Anniversary';
  spc CONSTANT VARCHAR2(1) := CHR(32); -- ASCII-код пробела
```

продолжение ➤

```

BEGIN
  display_title ('Happy Birthday');           -- Литерал
  display_title (happy_title);                -- Константа
  changing_title := happy_title;
  display_title (changing_title);             -- Переменная
  display_title ('Happy' || spc || 'Birthday'); -- Выражение
  display_title (INITCAP (happy_title));      -- Другое выражение
END;
```

А если вам потребуется передать данные из своей программы? В таком случае используйте параметр `OUT` или `IN OUT` — или рассмотрите возможность преобразования процедуры в функцию.

Режим OUT

Как вы, вероятно, уже поняли, параметр `OUT` по смыслу противоположен параметру `IN`. Параметры `OUT` могут использоваться для возвращения значений из программы вызывающему блоку PL/SQL. Параметр `OUT` сходен с возвращаемым значением функции, но он включается в список параметров, и количество таких параметров не ограничено (строго говоря, PL/SQL разрешает использовать до 64 000 параметров, но на практике это вряд ли можно считать ограничением).

В программе параметр `OUT` работает как неинициализированная переменная. Собственно, параметр `OUT` вообще не содержит никакого значения до успешного завершения вызванной программы (если только вы не использовали ключевое слово `NOCOPY`, которое подробно рассматривается в главе 21). Во время выполнения программы все операции присваивания параметру `OUT` в действительности выполняются с внутренней копией параметра. Когда программа успешно завершается и возвращает управление вызывающему блоку, значение локальной копии перемещается в параметр `OUT`. После этого значение становится доступным в вызывающем блоке PL/SQL.

У правил, относящихся к параметрам `OUT`, есть несколько практических следствий:

- Параметрам `OUT` нельзя задавать значения по умолчанию. Значение параметра `OUT` может задаваться только в теле модуля.
- Все операции присваивания параметрам `OUT` отменяются при инициировании исключения в программе. Так как значение параметра `OUT` присваивается только в случае успешного завершения программы, все промежуточные присваивания игнорируются. Если обработчик не перехватит исключение и не присвоит значение параметру `OUT`, параметр останется неизменным. Переменная сохранит значение, которое она имела до вызова программы.
- Фактический параметр, соответствующий формальному параметру `OUT`, не может быть константой, литералом или выражением. Иначе говоря, он должен поддерживать присваивание.

Как упоминается выше в таблице, Oracle *позволяет* прочитать значение параметра `OUT` в подпрограмме. Это значение изначально всегда равно `NULL`, но после его присваивания в подпрограмме оно становится «видимым», как показывает следующий сценарий:

```

SQL> set serveroutput on
SQL> CREATE OR REPLACE PROCEDURE read_out (n OUT NUMBER)
2  IS
3  BEGIN
4      DBMS_OUTPUT.put_line ('n initial=' || n);
5      n := 1;
6      DBMS_OUTPUT.put_line ('n after assignment=' || n);
7  END;
8  /
```


Procedure created.

```
SQL> DECLARE
  2     n    NUMBER;
  3 BEGIN
  4     read_out (n);
  5 END;
  6 /
n initial=
n after assignment=1
```

Режим IN OUT

В параметре IN OUT можно передавать значения программе и возвращать их на сторону вызова (либо исходное, неизменное значение, либо новое значение, заданное в программе). На параметры IN OUT распространяются два ограничения параметров OUT:

- Параметр IN OUT не может быть константой, литералом или выражением.
- Фактический параметр IN OUT должен быть переменной. Он не может быть константой, литералом или выражением, потому что эти форматы не могут использоваться PL/SQL в качестве приемника для размещения исходящих значений.

Других ограничений для параметров IN OUT нет.

Параметры IN OUT могут использоваться в обеих сторонах присваивания, потому что они работают как инициализированные переменные. PL/SQL не теряет значение параметра IN OUT в начале выполнения программы. Это значение может использоваться в программе там, где это необходимо.

Процедура `combine_and_format_names` объединяет имя и фамилию в заданном формате («LAST, FIRST» или «FIRST LAST»). Процедура получает имя и фамилию, которые преобразуются к верхнему регистру. В ней продемонстрированы все три режима параметров:

```
PROCEDURE combine_and_format_names
  (first_name_inout IN OUT VARCHAR2,
   last_name_inout  IN OUT VARCHAR2,
   full_name_out    OUT VARCHAR2,
   name_format_in   IN VARCHAR2 := 'LAST, FIRST')
IS
BEGIN
  /* Преобразование имени и фамилии к верхнему регистру. */
  first_name_inout := UPPER (first_name_inout);
  last_name_inout  := UPPER (last_name_inout);
  /* Объединение имени и фамилии в соответствии с форматной строкой. */
  IF name_format_in = 'LAST, FIRST'
  THEN
    full_name_out := last_name_inout || ', ' || first_name_inout;
  ELSIF name_format_in = 'FIRST LAST'
  THEN
    full_name_out := first_name_inout || ' ' || last_name_inout;
  END IF;
END combine_and_format_names;
```

Параметры имени и фамилии должны задаваться в режиме IN OUT. Параметр `full_name_out` должен быть параметром OUT, потому что процедура возвращает результат объединения имени и фамилии. Наконец, параметр `name_format_in`, содержащий форматную строку, объявляется в режиме IN, потому что он описывает способ форматирования, но никак не изменяется в процедуре.

Каждый режим параметров имеет свои характеристики и предназначение. Тщательно выбирайте режим, назначаемый вашим параметрам, чтобы они правильно использовались в модулях.



Определяйте формальные параметры в режимах OUT и IN OUT только в процедурах. Функции должны возвращать всю свою информацию исключительно командой RETURN. Выполнение этой рекомендации упростит понимание и использование ваших подпрограмм. Кроме того, функции с параметрами OUT и IN OUT не могут вызываться из команд SQL.

Связывание формальных и фактических параметров в PL/SQL

Каким образом при выполнении программы определяется, какому формальному параметру должен соответствовать фактический параметр? PL/SQL предлагает два метода установления такого соответствия:

- *по позиции* (неявное связывание);
- *по имени* (явное связывание с указанием имени формального параметра и обозначения =>).

Позиционное связывание

Во всех приводившихся ранее примерах применялось связывание параметров в соответствии с их позицией. При использовании этого способа PL/SQL принимает во внимание относительные позиции параметров, то есть N-й фактический параметр в вызове программы связывается с N-м формальным параметром в заголовке программы.

В следующем примере PL/SQL связывает первый фактический параметр `:order.company_id` с первым формальным параметром `company_id_in`, а второй фактический параметр `'N'` — со вторым формальным параметром `status_in`:

```
new_sales := total_sales (:order.company_id, 'N');
```

```
FUNCTION total_sales
  (company_id_in IN company.company_id%TYPE,
   status_in IN order.status_code%TYPE := NULL)
RETURN std_types.dollar_amount;
```

Метод позиционного связывания параметров, безусловно, является более наглядным и понятным (рис. 17.3).

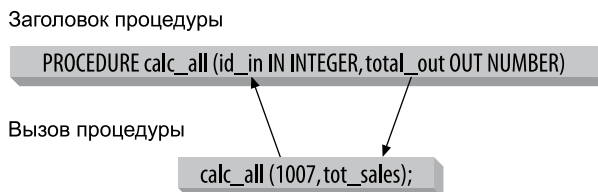


Рис. 17.3. Позиционное связывание параметров

Связывание по имени

Чтобы установить соответствие параметров по имени, следует при вызове подпрограммы явно связать формальный параметр с фактическим. Для этого используется комбинация символов `=>`:

```
имя_формального_параметра => значение_параметра
```

Поскольку имя формального параметра указано явно, PL/SQL уже не нужно учитывать порядок параметров. Поэтому в данном случае при вызове модуля не обязательно

перечислять параметры в порядке их следования в заголовке. Функцию `total_sales` можно вызывать любым из двух способов:

```
new_sales :=
  total_sales (company_id_in => order_pkg.company_id, status_in => 'N');
```

```
new_sales :=
  total_sales (status_in => 'N', company_id_in => order_pkg.company_id);
```

В одном вызове можно комбинировать оба метода связывания фактических и формальных параметров:

```
:order.new_sales := total_sales (order_pkg.company_id, status_in => 'N');
```

При этом все «позиционные» параметры должны быть перечислены перед «именованными», как в приведенном примере. Позиционному методу необходима начальная точка для отсчета позиций, которой может быть только первый параметр. Если разместить «именованные» параметры перед «позиционными», PL/SQL сойдет со счета. Оба вызова `total_sales`, приведенные ниже, недопустимы. В первой команде «именованные» параметры указываются до «позиционных», а во второй используется позиционная запись, но параметры перечислены в неверном порядке. В этом случае PL/SQL попытается преобразовать 'N' к типу `NUMBER` (для параметра `company_id`):

```
l_new_sales := total_sales (company_id_in => order_pkg.company_id, 'N');
l_new_sales := total_sales ('N', company_id_in => order_pkg.company_id);
```

Преимущества связывания по имени

Возникает резонный вопрос: зачем использовать связывание по имени? У него есть два основных преимущества:

- **Повышение информативности.** При использовании связывания по имени вызов программы содержит сведения о формальных параметрах, с которыми ассоциируются фактические значения. Если вы не знаете, что делают модули, вызываемые приложением, наличие имен формальных параметров помогает понять назначение конкретного программного вызова. В некоторых средах разработки именно по этой причине связывание по имени считается стандартным. Эта причина особенно актуальна в том случае, когда имена формальных параметров строятся по схеме с присоединением суффикса режима. Только взглянув на вызов процедуры или функции, разработчик сразу видит, в каком направлении передаются те или иные данные.
- **Гибкость при описании параметров.** Вы имеете возможность размещать параметры в удобном для работы порядке (конечно, это не значит, что при вызове аргументы можно перечислять так, как вам заблагорассудится!) и включать в список только те из них, которые действительно необходимы. В сложных приложениях иногда создаются процедуры с десятками параметров. Но как вы знаете, любой параметр, значение которого определяется по умолчанию, при вызове модуля может быть опущен. Разработчик может передать процедуре только те значения, которые необходимы для решения текущей задачи.

Давайте посмотрим, как реализовать все эти преимущества в программе. Возьмем для примера следующий заголовок:

```
/* Файл в Сети: namednot.sql */
PROCEDURE business_as_usual (
  advertising_budget_in   IN    NUMBER
  , contributions_inout    IN OUT NUMBER
  , merge_and_purge_on_in  IN    DATE DEFAULT SYSDATE
  , obscene_ceo_bonus_out  OUT   NUMBER
  , cut_corners_in         IN    VARCHAR2 DEFAULT 'WHENEVER POSSIBLE'
);
```

Анализ списка параметров приводит нас к следующим выводам:

- Минимальное количество аргументов, которые должны передаваться `business_as_usual`, равно 3. Чтобы определить его, сложите количество параметров `IN`, не имеющих значения по умолчанию, с количеством параметров `OUT` и `IN OUT`.
- При использовании связывания по позиции программа может вызываться с четырьмя или пятью аргументами, потому что последний параметр объявлен в режиме `IN` со значением по умолчанию.
- Для хранения значений, возвращаемых параметрами `OUT` и `IN OUT`, понадобятся как минимум две переменные.

С таким списком параметров программа может вызываться несколькими способами:

- Связывание только по позиции: задаются все фактические параметры. Обратите внимание, как трудно вспомнить, какой формальный параметр связывается с каждым из этих значений (и каков его смысл):

```
DECLARE
    l_ceo_payoff          NUMBER;
    l_lobbying_dollars    NUMBER := 100000;
BEGIN
    /* Позиционная запись */
    business_as_usual (50000000
                      , l_lobbying_dollars
                      , SYSDATE + 20
                      , l_ceo_payoff
                      , 'PAY OFF OSHA'
                      );
```

- Связывание только по позиции: минимальное количество фактических параметров. Понять смысл вызова все еще нелегко:

```
business_as_usual (50000000
                  , l_lobbying_dollars
                  , SYSDATE + 20
                  , l_ceo_payoff
                  );
```

- Связывание только по имени с сохранением исходного порядка. Обратите внимание: смысл вызова `business_as_usual` вполне понятен и не требует пояснений:

```
business_as_usual
(advertising_budget_in      => 50000000
 , contributions_inout      => l_lobbying_dollars
 , merge_and_purge_on_in   => SYSDATE
 , obscene_ceo_bonus_out    => l_ceo_payoff
 , cut_corners_in          => 'DISBAND OSHA'
);
```

- Все параметры `IN` пропускаются, и для них используются значения по умолчанию (другая важная возможность связывания по имени):

```
business_as_usual
(advertising_budget_in      => 50000000
 , contributions_inout      => l_lobbying_dollars
 , obscene_ceo_bonus_out    => l_ceo_payoff
);
```

- Связывание по имени с изменением порядка следования фактических параметров и передачей неполного списка:

```
business_as_usual
(obscene_ceo_bonus_out      => l_ceo_payoff
 , merge_and_purge_on_in    => SYSDATE
 , advertising_budget_in     => 50000000
 , contributions_inout       => l_lobbying_dollars
);
```

- Объединение связывания по имени и по позиции. Список параметров начинается со связывания по позиции, а после перехода на связывание по имени вернуться к позиционному связыванию уже не удастся:

```
business_as_usual
(50000000
 , l_lobbying_dollars
 , merge_and_purge_on_in      => SYSDATE
 , obscene_ceo_bonus_out      => l_ceo_payoff
 );
```

Как видите, механизм передачи аргументов в PL/SQL весьма гибок. Как правило, связывание по имени способствует написанию кода, который лучше читается и создает меньше проблем с сопровождением. От вас потребуется совсем немного: найти и записать имена параметров.

Квалификатор режима параметра NOCOPY

PL/SQL предоставляет возможность изменить определение параметра при помощи конструкции `NOCOPY`. `NOCOPY` требует, чтобы компилятор PL/SQL не создавал копии аргументов `OUT` и `IN` — и в большинстве случаев компилятор это требование удовлетворяет. `NOCOPY` используется прежде всего для повышения эффективности передачи больших конструкций (например, коллекций) в аргументах `IN OUT`. Эта тема напрямую связана с производительностью, поэтому она более подробно рассматривается в главе 21.

Значения по умолчанию

Как было показано в предыдущих примерах, параметрам `IN` можно задать значения по умолчанию. Если параметр `IN` имеет значение по умолчанию, включать этот параметр в вызов программы не обязательно. Значение по умолчанию параметра вычисляется и используется программой только в том случае, если параметр не был включен в список при вызове. Конечно, для всех параметров `IN OUT` фактические параметры должны включаться в список. Значение по умолчанию определяется для параметра так же, как для объявляемой переменной. Предусмотрены два способа задания значений по умолчанию — с ключевым словом `DEFAULT` и с оператором присваивания (`:=`):

```
PROCEDURE astrology_reading
(sign_in IN VARCHAR2 := 'LIBRA',
 born_at_in IN DATE DEFAULT SYSDATE) IS
```

Значения по умолчанию позволяют вызывать программы с разным количеством фактических параметров. Программа использует значения по умолчанию для всех незадаанных параметров, а для параметров в списке значения по умолчанию заменяются указанными значениями. Несколько примеров разных вариантов использования связывания по позиции:

```
BEGIN
  astrology_reading ('SCORPIO',
    TO_DATE ('12-24-2009 17:56:10', 'MM-DD-YYYY HH24:MI:SS'));
  astrology_reading ('SCORPIO');
  astrology_reading;
  astrology_reading();
END;
```

В первом вызове оба параметра задаются явно. Во втором вызове задается только первый фактический параметр, поэтому `born_at_in` присваивается текущая дата и время. В третьем вызове параметры вообще не указаны, поэтому круглые скобки отсутствуют (то же относится и к последнему вызову, в который включены пустые круглые скобки). Оба значения по умолчанию используются в теле процедуры.

Чтобы пропустить начальные параметры, имеющие значения по умолчанию, необходимо использовать связывание по имени. Включая имена формальных параметров, можно указать только те параметры, для которых необходимо передать значения:

```
BEGIN
    astrology_reading (
        born_at_in =>
            TO_DATE ('12-24-2009 17:56:10', 'MM-DD-YYYY HH24:MI:SS'));
END;
```

Локальные модули

Локальный (или вложенный) модуль — это процедура или функция, определяемая в разделе объявлений блока PL/SQL (анонимного или именованного). Локальным такой модуль называется из-за того, что он определяется только внутри родительского блока PL/SQL и не может быть вызван из другого блока, определенного вне родительского. Как показано на рис. 17.4, блоки, внешние по отношению к определению процедуры, не могут непосредственно вызывать ее локальные процедуры или функции.

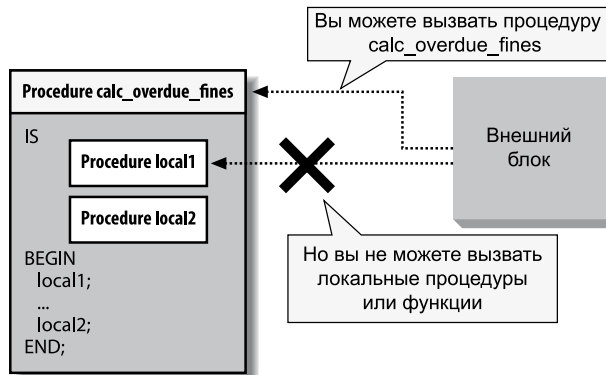


Рис. 17.4. Локальные модули недоступны вне процедуры

Синтаксис определения процедуры или функции не отличается от синтаксиса создания отдельных модулей. Например, в приведенном ниже анонимном блоке объявлена локальная процедура `show_date`:

```
DECLARE
    PROCEDURE show_date (date_in IN DATE) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE (TO_CHAR (date_in, 'Month DD, YYYY'));
    END show_date;
BEGIN
    ...
END ;
```

В разделе объявлений локальные модули должны располагаться после остальных объявлений. Таким образом, переменные, курсоры, исключения, типы, записи и т. д. должны объявляться до первого ключевого слова `PROCEDURE` или `FUNCTION`.

Преимущества локальных модулей

Использование локальных модулей обеспечивает следующие преимущества:

- **Уменьшение объема кода за счет исключения повторяющихся фрагментов.** Самая распространенная причина для создания локальных модулей. Уменьшение объема

кода сопровождается повышением его качества, так как число строк и количество потенциальных ошибок в них уменьшаются, а поддержка кода значительно упрощается. Изменения вносятся только в локальном модуле, а результаты немедленно проявляются в родительском модуле.

- **Улучшение удобочитаемости кода.** Даже если модуль не содержит повторяющихся блоков кода, взаимосвязанные операторы лучше поместить в локальный модуль. В этом случае вам будет легче отслеживать логику родительского модуля.

Уменьшение объема кода

Рассмотрим пример уменьшения объема кода. Процедура `calc_percentages` получает числовые значения из пакета `sales` (`sales_pkg`), вычисляет процент отдельных продаж для общего объема продаж, передаваемого в параметре, после чего форматирует число для вывода в отчете или на форме. В следующем примере задействованы всего три вычисления, но я извлек его из коммерческого приложения, в котором их было 23!

```
PROCEDURE calc_percentages (total_sales_in IN NUMBER)
IS
  l_profile sales_descriptors%ROWTYPE;
BEGIN
  l_profile.food_sales_stg :=
    TO_CHAR ((sales_pkg.food_sales / total_sales_in ) * 100,
             '$999,999');
  l_profile.service_sales_stg :=
    TO_CHAR ((sales_pkg.service_sales / total_sales_in ) * 100,
             '$999,999');
  l_profile.toy_sales_stg :=
    TO_CHAR ((sales_pkg.toy_sales / total_sales_in ) * 100,
             '$999,999');
END;
```

Этот код (относительно) долго пишется, занимает больше места, чем необходимо, и создает проблемы при сопровождении. А если понадобится изменить формат, к которому преобразуются числа? А если изменится формула вычислений? Придется вносить изменения во все вычисления.

С локальными модулями я могу выделить общий, повторяющийся код в функцию, которая будет многократно вызываться в `calc_percentages`. Версия локального модуля этой процедуры выглядит так:

```
PROCEDURE calc_percentages (total_sales_in IN NUMBER)
IS
  l_profile sales_descriptors%ROWTYPE;
  /* Функция определяется прямо в процедуре! */
  FUNCTION pct_stg (val_in IN NUMBER) RETURN VARCHAR2
  IS
  BEGIN
    RETURN TO_CHAR ((val_in/total_sales_in ) * 100, '$999,999');
  END;
BEGIN
  l_profile.food_sales_stg := pct_stg (sales_pkg.food_sales);
  l_profile.service_sales_stg := pct_stg (sales_pkg.service_sales);
  l_profile.toy_sales_stg := pct_stg (sales_pkg.toy_sales);
END;
```

Все сложности вычисления, от деления на `total_sales_in` до умножения на 100 и форматирования функцией `TO_CHAR`, были выделены в функцию `pct_stg`. Эта функция определяется в разделе объявлений в процедуре. Вызов этой функции из тела `calc_percentages` упрощает чтение и сопровождение команд процедуры. Если формула вычислений изменится, то изменения достаточно внести всего в одной функции, а они отразятся на всех операциях присваивания.

Улучшение удобочитаемости кода

Локальные модули способны кардинально упростить чтение и сопровождение кода. По сути, локальные модули позволяют очень близко воспроизвести методологию нисходящего проектирования или пошаговой проработки. Те же приемы могут использоваться для декомпозиции или рефакторинга существующих программ, чтобы упростить их чтение.

Конечным результатом такого использования локальных модулей является радикальное уменьшение исполняемых разделов (многие строки логики перемещаются из исполняемого раздела в локальный модуль, вызываемый в этом разделе). С малым размером исполняемых разделов такая логика гораздо проще читается и становится более понятной.



Я рекомендую включить в ваши стандарты программирования ограничение, согласно которому длина исполняемых разделов в блоках PL/SQL не превышает 60 строк (объем текста, помещающегося на экране или странице). Это может показаться странным, однако соблюдение этих рекомендаций не только возможно, но и очень полезно.

Предположим, программа содержит серию циклов `WHILE` (в том числе и вложенных), тела которых содержат серию сложных вычислений и глубоко вложенную условную логику. Даже с обширными комментариями разобраться в логике программы, разнесенной на несколько страниц, будет нелегко — особенно если завершители `END IF` и `END LOOP` находятся на разных страницах с открывающими командами `IF` и `LOOP`.

Напротив, если извлечь последовательность взаимосвязанных команд, разместить их в одном или нескольких локальных модулях, а затем вызвать эти модули в теле программы, такая программа буквально будет документировать сама себя. Процедура `assign_workload` предоставляет упрощенную версию этого сценария, которая наглядно демонстрирует преимущества локальных модулей:

```
/* Файл в Сети: local_modules.sql */
PROCEDURE assign_workload (department_in IN emp.deptno%TYPE)
IS
    CURSOR emps_in_dept_cur (department_in IN emp.deptno%TYPE)
    IS
        SELECT * FROM emp WHERE deptno = department_in;

    PROCEDURE assign_next_open_case
        (emp_id_in IN NUMBER, case_out OUT NUMBER)
    IS
    BEGIN ... полная реализация ... END;

    FUNCTION next_appointment (case_id_in IN NUMBER)
        RETURN DATE
    IS
    BEGIN ... полная реализация ... END;

    PROCEDURE schedule_case
        (case_in IN NUMBER, date_in IN DATE)
    IS
    BEGIN ... полная реализация ... END;
BEGIN /* main */
    FOR emp_rec IN emps_in_dept_cur (department_in)
    LOOP
        IF analysis.caseload (emp_rec.emp_id) <
            analysis.avg_cases (department_in);
        THEN
            assign_next_open_case (emp_rec.emp_id, case#);
            schedule_case
```



```
        (case#, next_appointment (case#));  
    END IF;  
END LOOP  
END assign_workload;
```

Процедура `assign_workload` состоит из трех локальных модулей:

```
assign_next_open_case  
next_appointment  
schedule_case
```

Она также зависит от двух пакетных программ, которые уже существуют и легко подключаются к этой программе: `analysis.caseload` и `analysis.avg_cases`. Для понимания логики, лежащей в основе `assign_workload`, неважно, какой код выполняется в каждой из них. Я могу просто положиться на имена этих модулей в процессе чтения основного кода. Даже без комментариев читатель кода будет четко представлять, что делает каждый модуль. Конечно, если вы предпочитаете обеспечивать самодокументирование кода на уровне именованных объектов, вам придется придумать очень содержательные имена для ваших функций и процедур.

Область действия локальных модулей

Модульный раздел объявлений похож на тело пакета (см. главу 18). Тело пакета тоже содержит определения модулей. Главное различие между локальными и пакетными модулями связано с их областью действия. Локальные модули могут вызываться только из блока, в котором они определяются; пакетные модули могут (как минимум!) вызываться из любой точки пакета. Если пакетные модули также включены в спецификацию пакета, они могут вызываться другими программными единицами из схем, имеющих привилегии `EXECUTE` для этого пакета.

Следовательно, локальные модули следует применять только для инкапсуляции кода, который не нужно вызывать за пределами текущей программы. А если нужно — создайте пакет!

Вложенные подпрограммы

Каждый раз, когда я пишу сколько-нибудь нетривиальную программу длиной более 20 строк, я создаю один или несколько локальных модулей. Это помогает мне следить за логикой решения; я могу рассматривать свой код на более высоком уровне абстракции, присваивая имя целой последовательности команд, а также выполнять нисходящее проектирование и пошаговую проработку требований. Наконец, модуляризация кода даже в рамках одной программы позволяет легко выделить вложенную подпрограмму и создать действительно автономную процедуру или функцию, пригодную для повторного использования.

Конечно, логику также можно вывести из локальной области действия и преобразовать ее в отдельную программу уровня тела пакета (если предположить, что код пишется в составе пакета). Этот подход сокращает глубину вложения локальных процедур, что может быть полезно. Однако он может привести к появлению тел пакетов, содержащих очень длинные списки программ, многие из которых используются только из другой программы. Мой общий принцип заключается в том, что определение элемента должно располагаться как можно ближе к месту его использования, что естественным образом приводит к созданию вложенных подпрограмм.

Надеюсь, сейчас вам вспомнилась какая-нибудь из написанных вами программ. Вернитесь к ней и устраните повторяющийся код, почистите логику — в общем, сделайте

программу более понятной для другого читателя. Не боритесь с искушением. Проведите модуляризацию своего кода.

Чтобы помочь вам в определении вложенных подпрограмм и работе с ними в ваших приложениях, я создал пакет **TopDown**. При помощи этого пакета вы тратите немного времени на расстановку в коде «индикаторов» (фактически инструкций относительно того, какие вложенные подпрограммы вы хотите построить и как именно). Затем вы компилируете этот шаблон в базу данных, вызываете **TopDown.Refactor** для программного модуля — и вложенные подпрограммы строятся автоматически!

Процесс можно повторить на каждом последующем уровне структуры вашей программы. При этом быстро формируется модульная архитектура, которую вы (и ваши коллеги) непременно оцените в будущем.

Более полное описание пакета **TopDown**, исходный код и примеры сценариев приводятся в файле **TopDown.zip** на сайте книги.

Перегрузка подпрограмм

Программы, которые существуют в одной области видимости и имеют одинаковые имена, называются *перегруженными*. PL/SQL поддерживает перегрузку процедур и функций в разделе объявлений блока (именованного или неименованного), в теле и спецификации пакета, а также в объявлении объектного типа. Перегрузка — это очень мощный механизм, и вы должны научиться использовать все его возможности.

В следующем простом примере представлены три перегруженные подпрограммы, определенные в разделе объявлений анонимного блока (а следовательно, являющихся локальными):

```
DECLARE
  /* Первая версия получает параметр DATE. */
  FUNCTION value_ok (date_in IN DATE) RETURN BOOLEAN IS
  BEGIN
    RETURN date_in <= SYSDATE;
  END;

  /* Вторая версия получает параметр NUMBER. */
  FUNCTION value_ok (number_in IN NUMBER) RETURN BOOLEAN IS
  BEGIN
    RETURN number_in > 0;
  END;

  /* Третья версия - процедура! */
  PROCEDURE value_ok (number_in IN NUMBER) IS
  BEGIN
    IF number_in > 0 THEN
      DBMS_OUTPUT.PUT_LINE (number_in || 'is OK!');
    ELSE
      DBMS_OUTPUT.PUT_LINE (number_in || 'is not OK!');
    END IF;
  END;
END;
```

BEGIN

Когда исполняемое ядро PL/SQL встречает в программе строку вида

```
IF value_ok (SYSDATE) THEN ...
```

список фактических параметров сравнивается со списками формальных параметров разных перегруженных модулей. В результате PL/SQL выполняет код того модуля, для которого указанные списки совпадают.



Перегрузку также называют статическим полиморфизмом. Термином «полиморфизм» обозначается возможность языка определять и использовать несколько программ с одинаковыми именами. Если вызываемый вариант определяется на стадии компиляции — это статический полиморфизм. Если решение принимается во время выполнения, то используется термин «динамический полиморфизм»; этот вид полиморфизма обеспечивается наследованием объектных типов.

Перегрузка способна сильно упростить жизнь как вам, так и другим разработчикам. Она консолидирует интерфейсы вызова многих похожих программ в одно имя модуля. «Информационное бремя» перекладывается с разработчика на программный продукт. Например, вам не придется пытаться запомнить шесть разных имен для программ, добавляющих значения (даты, строки, флаги, числа и т. д.) в разные коллекции. Вместо этого вы просто сообщаете компилятору, что вы хотите добавить значение, и передаете это значение. PL/SQL и перегруженные программы определяют, что вы хотите сделать, и выполняют необходимые действия за вас.

При создании перегруженных подпрограмм вы потратите больше времени на проектирование и реализацию, чем с отдельными, автономными программами. Дополнительное время с лихвой окупится в будущем, потому что и вам, и другим разработчикам будет проще работать с вашим кодом.

Преимущества перегрузки

Перегрузка модулей наиболее уместна в следующих ситуациях:

- **Необходимость поддержки разных типов и наборов данных.** Если одно и то же действие применяется к разным видам и наборам данных, перегрузка обеспечивает разные варианты активизации этого действия, а не просто позволяет называть разные действия одним именем.
- **Адаптация программы под разные требования.** Чтобы ваш код получился по возможности универсальным, можно создать несколько его версий с разными вариантами вызова. Для этого часто требуется выполнять перегрузку процедур и функций. Хорошим признаком ее уместности служит невостребованный, «лишний» код. К примеру, работая с пакетом `DBMS_SQL`, вы вызываете функцию `DBMS_SQL.EXECUTE`, но при выполнении ею DDL-команд возвращаемое значение игнорируется. Если для этой функции создать перегруженную процедуру, DDL-команду можно выполнить следующим образом:

```
BEGIN
  DBMS_SQL.EXECUTE ('CREATE TABLE xyz ...');
```

Без применения перегрузки вы должны вызвать функцию

```
DECLARE
  feedback PLS_INTEGER;
BEGIN
  feedback := DBMS_SQL.EXECUTE ('CREATE TABLE xyz ...');
```

а затем игнорировать переменную `feedback`.

- **Необходимость перегрузки по типу, а не по значению.** Самая редкая причина для использования перегрузки. В этом случае версия перегруженной программы выбирается в зависимости от типа данных, а не их значения. Хорошим примером такого рода служит функция `DBMS_SQL.DEFINE_COLUMN`: пакету `DBMS_SQL` необходимо сообщить тип каждого столбца, выбранного при помощи динамического запроса. Для определения числового столбца можно использовать вызов

```
DBMS_SQL.DEFINE_COLUMN (cur, 1, 1);
```

ИЛИ ВЫЗОВ

```
DBMS_SQL.DEFINE_COLUMN (cur, 1, DBMS_UTILITY.GET_TIME);
```

Конкретное значение роли не играет; мы должны указать «это число», а не привести какое-то конкретное число. Перегрузка обеспечивает элегантное решение этой проблемы.

Сейчас мы рассмотрим типичный пример перегрузки, а затем разберемся с ограничениями и рекомендациями по поводу перегрузки.

Поддержка разных комбинаций данных

Перегрузка может использоваться для выполнения одного действия с разными комбинациями данных. Как упоминалось ранее, такая разновидность перегрузки предоставляет не столько общее имя для разных операций, как разные способы вызова одной операции. Возьмем `DBMS_OUTPUT.PUT_LINE`: эта встроенная функция может использоваться для вывода значений любых типов данных, которые могут быть явно или неявно преобразованы в строку. Интересно, что в предшествующих версиях Oracle Database (7, 8, 8i, 9i) эта процедура была перегружена, но в Oracle Database 10g и выше она не перегружается! Таким образом, если вам потребуется вывести выражение, которое не может быть неявно преобразовано в строку, вы не сможете вызвать `DBMS_OUTPUT.PUT_LINE` и передать это выражение.

И что из того, спросите вы? PL/SQL неявно преобразует числа и даты в строки. Какие еще данные приходится выводить? Для начала — как насчет логических значений? Чтобы вывести значение логического выражения, придется написать команду `IF` следующего вида:

```
IF l_student_is_registered
THEN
    DBMS_OUTPUT.PUT_LINE ('TRUE');
ELSE
    DBMS_OUTPUT.PUT_LINE ('FALSE');
END IF;
```

Глупо, вам не кажется? И разве это не напрасная потеря времени? К счастью, проблема решается очень просто. Постройте на базе `DBMS_OUTPUT.PUT_LINE` свой пакет с множеством перегрузок. Ниже приводится сильно сокращенная версия такого пакета. При желании вы сможете легко расширить ее, как я делаю в процедуре `do.pl`. Часть спецификации пакета выглядит так:

```
/* Файл в Сети: do.pkg (также см. файлы p.* ) */
PACKAGE do
IS
    PROCEDURE pl (boolean_in IN BOOLEAN);
    /* Вывод строки. */
    PROCEDURE pl (char_in IN VARCHAR2);
    /* Вывод строки и значения BOOLEAN. */
    PROCEDURE pl (
        char_in      IN   VARCHAR2,
        boolean_in   IN   BOOLEAN
    );
    PROCEDURE pl (xml_in IN SYS.XMLType);
END do;
```

Пакет просто расширяет `DBMS_OUTPUT.PUT_LINE`. При использовании `do.pl` я могу вывести логическое значение без написания команды `IF`, как в следующем примере:

```
DECLARE
    v_is_valid BOOLEAN :=
        book_info.is_valid_isbn ('5-88888-66');
BEGIN
    do.pl (v_is_valid);
```

Более того, `do.pl` можно применять даже к сложным типам данных — таким, как `XMLType`:

```
/* Файл в Сети: xmltype.sql */  
DECLARE  
  doc  xmltype;  
BEGIN  
  SELECT ea.report  
  INTO  doc  
  FROM  env_analysis ea  
  WHERE company= 'ACME SILVERPLATING';  
  do.pl (doc);  
END;
```

Ограничения на использование перегрузки

При выполнении перегрузки необходимо учитывать несколько обстоятельств. Исполнительное ядро PL/SQL как на этапе компиляции, так и на этапе запуска должно быть в состоянии отличить друг от друга перегруженные версии подпрограммы; ведь оно не может выполнять две подпрограммы одновременно. Так как все перегруженные версии имеют одинаковые имена, исполнительное ядро не может различать их по именам. Для определения того, какую из версий следует выполнить, PL/SQL использует списки параметров и/или сведения о типе программы (процедура или функция). Все это приводит к тому, что на перегруженные программы накладывается ряд ограничений.

У перегруженных программ типы хотя бы одного параметра должны принадлежать к разным семействам. Типы данных `INTEGER`, `REAL`, `DECIMAL`, `FLOAT` и т. д. являются числовыми подтипами, а типы `CHAR`, `VARCHAR2` и `LONG` — символьными. Если соответствующие параметры отличаются только подтипом, то есть принадлежат к одному супертипу, у PL/SQL не будет достаточной информации для выбора выполняемой программы.



В следующем разделе описаны некоторые усовершенствования Oracle10g (и последующих версий), относящиеся к перегрузке числовых типов.

- Если у перегруженных программ различаются только имена параметров, то при вызове таких программ должно использоваться связывание по имени. Если имя аргумента не указано, как компилятор сможет различить вызовы двух перегруженных программ? И все же ситуаций, в которых связывание по имени отличается по семантическому смыслу от позиционной записи, следует по возможности избегать.
- Списки параметров перегруженных программ не должны различаться только режимом использования параметров. Даже если в одной версии для параметра задан режим `IN`, а в другой — `IN OUT`, PL/SQL не видит различия между ними.
- Все перегруженные программы должны объявляться в одном и том же блоке PL/SQL или иметь одну область видимости (анонимный блок, пакет, отдельная процедура или функция). Нельзя определить одну версию программы в одном блоке (с его областью видимости), а другую — в другом. Вы не можете перегружать две отдельные программы, так как в этом случае одна из них просто заменит другую.
- Перегруженные функции не могут различаться только типом возвращаемого значения (указанного в предложении `RETURN` функции). В момент вызова перегруженной функции компилятор не знает, значение какого типа она будет возвращать. Поэтому он не может определить, какую из версий функции использовать, если все остальные параметры идентичны.

Перегрузка числовых типов

В Oracle10g появилась возможность перегрузки двух подпрограмм, различающихся только числовым типом формальных параметров. Рассмотрим конкретный пример:

```
DECLARE
  PROCEDURE proc1 (n IN PLS_INTEGER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('pls_integer version');
  END;

  PROCEDURE proc1 (n IN NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('number version');
  END;
BEGIN
  proc1 (1.1);
  proc1 (1);
END;
```

При попытке выполнить этот код в Oracle9i происходит ошибка:

```
ORA-06550: line 14, column 4:
PLS-00307: too many declarations of 'PROC1' match this call
```

Однако при выполнении того же блока в Oracle10g и Oracle11g результат будет таким:

```
number version
pls_integer version
```

Теперь компилятор PL/SQL различает два вызова. Обратите внимание: при передаче не-целочисленного значения вызывается «NUMBER-версия» программы. Это объясняется тем, что PL/SQL при определении соответствия перебирает типы в следующем порядке: сначала PLS_INTEGER или BINARY_INTEGER, затем NUMBER, BINARY_FLOAT и наконец, BINARY_DOUBLE. Используется первая перегруженная программа, которая соответствует переданным значениям фактических аргументов.

Хотя эта новая гибкая возможность чрезвычайно удобна, будьте осторожны при использовании этой крайне неочевидной перегрузки — убедитесь в том, что она работает именно так, как предполагалось. Протестируйте свой код с разными входными данными и проверьте результаты. Помните, что в качестве числового параметра может быть передана строка вида «156.4»; протестируйте и такие данные.

Также можно уточнить тип числовых значений и использовать функции преобразования для явного выбора перегруженной версии. Скажем, если значение 5.0 должно передаваться в формате BINARY_FLOAT, используйте обозначение 5.0f или функцию преобразования TO_BINARY_FLOAT(5.0).

Опережающие объявления

Язык PL/SQL требует, чтобы элементы были объявлены до того, как они будут использоваться в коде программы. Но как PL/SQL определит, насколько корректно они применяются? Если одни модули вызывают другие, может возникнуть ситуация, когда невозможно объявить все модули до первой ссылки на них. Например, что делать, если программа А вызывает программу Б, а программа Б, в свою очередь, — программу А? К счастью, PL/SQL поддерживает *рекурсию*, в том числе и взаимную, при которой две и более программ явно либо неявно вызывают друг друга.

Хорошая новость для программистов, использующих взаимную рекурсию: PL/SQL позволяет выполнять *опережающее объявление* локальных модулей, то есть объявлять

модули до их фактической реализации. Это дает возможность вызывать модуль из других программ до его определения. Опережающее объявление состоит из заголовка программы, за которым следует точка с запятой. Эта конструкция называется *заголовком модуля*. Такой заголовок включает список параметров (и секцию `RETURN` — для функций) и предоставляет всю информацию, которая необходима PL/SQL для объявления модуля и правильной обработки любых обращений к нему.

В следующем примере внутри процедуры определяются две взаимно рекурсивные функции. Следовательно, заголовок второй функции `total_cost` должен определяться до объявления функции `net_profit`:

```
PROCEDURE perform_calcs (year_in IN INTEGER)
IS
  /* Только заголовок для функции total_cost. */
  FUNCTION total_cost (...) RETURN NUMBER;

  /* Функция net_profit использует total_cost. */
  FUNCTION net_profit (...) RETURN NUMBER IS
  BEGIN
    RETURN total_sales (...) - total_cost (...);
  END;

  /* Функция total_cost использует net_profit. */
  FUNCTION total_cost (...) RETURN NUMBER IS
  BEGIN
    IF <условие, основанное на параметрах>
    THEN
      RETURN net_profit (...) * .10;
    ELSE
      RETURN <значение параметра>;
    END IF;
  END;
END;
```

При использовании опережающих объявлений следует помнить некоторые правила:

- Программа не может содержать опережающие объявления переменных или курсоров. Этот прием работает только с модулями (процедурами или функциями).
- Определение программы должно содержаться в разделе объявлений того же блока PL/SQL (анонимного блока, процедуры, функции или тела пакета), в котором размещается опережающее объявление.

В одних случаях без опережающих объявлений не обойтись; в других они просто делают код более понятным и доступным. Опережающие объявления, как и все расширенные и нестандартные возможности PL/SQL, должны использоваться только тогда, когда они действительно необходимы.

Дополнительные вопросы

Этот раздел пригодится в основном опытным программистам, поскольку в нем рассматриваются нетривиальные аспекты модуляризации, в частности вызов функций в SQL и детерминированные функции.

Вызов пользовательских функций в SQL

Oracle позволяет вызывать пользовательские функции в коде SQL. Фактически это позволяет адаптировать язык SQL под требования конкретных приложений.



Каждый раз, когда исполнительное ядро SQL вызывает функцию PL/SQL, оно должно «переключаться» на исполнительное ядро PL/SQL. При многократном вызове функции затраты на переключение контекстов могут быть весьма значительными.

Требования к вызываемым функциям

Чтобы определяемую программистом функцию PL/SQL можно было вызывать из команд SQL, она должна отвечать следующим требованиям:

- Все параметры функции должны иметь режим использования IN. Режимы IN OUT и OUT в функциях, встраиваемых в SQL-код, недопустимы.
- Типы данных параметров функций и тип возвращаемого значения должны распознаваться сервером Oracle. PL/SQL дополняет основные типы Oracle, которые пока не поддерживаются базой данных. Речь идет о типах BOOLEAN, BINARY_INTEGER, ассоциативных массивах, записях PL/SQL и определяемых программистом подтипах.
- Функция должна храниться в базе данных. Функция, определенная на стороне клиента, не может вызываться в командах SQL, так как SQL не сможет разрешить ссылку на эту функцию.



По умолчанию пользовательские функции, вызываемые в SQL, оперируют данными одной строки, а не столбца (как агрегатные функции SUM, MIN и AVG). Чтобы создать агрегатные функции, вызываемые в SQL, необходимо использовать интерфейс ODCIAggregate, который является частью среды Oracle Extensibility Framework. За подробной информацией по этой теме обращайтесь к документации Oracle.

Ограничения для пользовательских функций, вызываемых в SQL

С целью защиты от побочных эффектов и непредсказуемого поведения хранимых процедур Oracle не позволяет им выполнять следующие действия:

- Хранимые функции не могут модифицировать таблицы баз данных и выполнять команды DDL (CREATE TABLE, DROP INDEX и т. д.), INSERT, DELETE, MERGE и UPDATE. Эти ограничения ослабляются, если функция определена как автономная транзакция (см. главу 14). В таком случае любые вносимые ею изменения осуществляются независимо от внешней транзакции, в которой выполняется запрос.
- Хранимые функции, которые вызываются удаленно или в параллельном режиме, не могут читать или изменять значения переменных пакета. Сервер Oracle не поддерживает побочные эффекты, действие которых выходит за рамки сеанса пользователя.
- Хранимая функция может изменять значения переменных пакета, только если она вызывается в списке выборки либо в предложении VALUES или SET. Если хранимая функция вызывается в предложении WHERE или GROUP BY, она не может изменять значения переменных пакета.
- До выхода Oracle8 пользовательские функции не могли вызывать процедуру RAISE_APPLICATION_ERROR.
- Хранимая функция не может вызывать другой модуль (хранимую процедуру или функцию), не соответствующий приведенным требованиям.
- Хранимая функция не может обращаться к представлению, которое нарушает любое из предшествующих правил. Представлением (view) называется хранимая команда SELECT, в которой могут вызываться хранимые функции.
- До выхода Oracle11g для передачи параметров функциям могла использоваться только позиционная запись. Начиная с Oracle11g, допускается передача параметров по имени и смешанная запись.

Непротиворечивость чтения и пользовательские функции

Модель непротиворечивости чтения в базе данных Oracle проста и понятна: после выполнения запрос «видит» данные в том состоянии, в котором они существовали (были зафиксированы в базе данных) на момент начала запроса, с учетом результатов изменений, вносимых командами DML текущей транзакции. Таким образом, если мой запрос был выполнен в 9:00 и продолжает работать в течение часа, даже если за это время другой пользователь внесет в данные изменения, они не отразятся в моем запросе.

Но если не принять специальных мер предосторожности с пользовательскими функциями в ваших запросах, может оказаться, что ваш запрос будет нарушать (по крайней мере на первый взгляд) модель непротиворечивости чтения базы данных Oracle. Чтобы понять этот аспект, рассмотрим следующую функцию и запрос, в котором она вызывается:

```
FUNCTION total_sales (id_in IN account.account_id%TYPE)
RETURN NUMBER
IS
  CURSOR tot_cur
  IS
    SELECT SUM (sales) total
      FROM orders
     WHERE account_id = id_in
       AND TO_CHAR (ordered_on, 'YYYY') = TO_CHAR (SYSDATE, 'YYYY');
  tot_rec tot_cur%ROWTYPE;
BEGIN
  OPEN tot_cur;
  FETCH tot_cur INTO tot_rec;
  CLOSE tot_cur;
  RETURN tot_rec.total;
END;
SELECT name, total_sales (account_id)
  FROM account
 WHERE status = 'ACTIVE';
```

Таблица `account` содержит 5 миллионов активных строк, а таблица `orders` — 20 миллионов. Я запускаю запрос в 10:00, на его завершение уходит около часа. В 10:45 приходит некто, обладающий необходимыми привилегиями, удаляет все строки из таблицы `orders` и закрепляет транзакцию. По правилам модели непротиворечивости чтения Oracle сеанс, в котором выполняется запрос, не должен рассматривать эти строки как удаленные до завершения запроса. Но при следующем вызове из запроса функция `total_sales` не найдет ни одной строки и вернет `NULL` — и так будет происходить до завершения запроса.

При выполнении запросов из функций, вызываемых в коде SQL, необходимо внимательно следить за непротиворечивостью чтения. Если эти функции вызываются в продолжительных запросах или транзакциях, вероятно, вам стоит выполнить следующую команду для обеспечения непротиворечивости чтения между командами SQL текущей транзакции:

```
SET TRANSACTION READ ONLY
```

В этом случае необходимо позаботиться о наличии достаточного табличного пространства отмены.

Определение подпрограмм PL/SQL в командах SQL (12.1 и выше)

Разработчики уже давно могли вызывать свои функции PL/SQL из команд SQL. Допустим, я создал функцию с именем `BETWNSTR`, которая возвращает подстроку с заданной начальной и конечной позицией:

```
FUNCTION betwnstr (
  string_in      IN   VARCHAR2
  , start_in     IN   PLS_INTEGER
  , end_in       IN   PLS_INTEGER
)
```

продолжение ➤

```

RETURN VARCHAR2
IS
BEGIN
RETURN ( SUBSTR (
    string_in, start_in, end_in - start_in + 1 ));
END;
```

Функция может использоваться в запросах следующим образом:

```

SELECT betwnstr (last_name, 3, 5)
FROM employees
```

Эта возможность позволяет «расширить» язык SQL функциональностью, присущей конкретному приложению, и повторно использовать алгоритмы (вместо копирования). К недостаткам выполнения пользовательских функций в SQL следует отнести необходимость переключения контекста между исполнительными ядрами SQL и PL/SQL. Начиная с Oracle Database 12c вы можете определять функции и процедуры PL/SQL в секции WITH подзапроса, чтобы затем использовать их как любую встроенную или пользовательскую функцию. Эта возможность позволяет консолидировать функцию и запрос в одной команде:

```

WITH
FUNCTION betwnstr (
    string_in  IN VARCHAR2,
    start_in   IN PLS_INTEGER,
    end_in     IN PLS_INTEGER)
RETURN VARCHAR2
IS
BEGIN
RETURN (SUBSTR (
    string_in,
    start_in,
    end_in - start_in + 1));
END;
SELECT betwnstr (last_name, 3, 5)
FROM employees
```

Главное преимущество такого решения — повышение производительности, так как при таком определении функций затраты на переключение контекста с ядра SQL на ядро PL/SQL существенно снижаются. С другой стороны, за него приходится расплачиваться возможностью повторного использования логики в других частях приложения.

Впрочем, для определения функций в секции WITH есть и другие причины. В SQL можно вызвать пакетную функцию, но нельзя сослаться на константу, объявленную в пакете (если только команда SQL не выполняется внутри блока PL/SQL), как показано в следующем примере:

```

SQL> CREATE OR REPLACE PACKAGE pkg
2  IS
3      year_number  CONSTANT INTEGER := 2013;
4  END;
5  /
```

Package created.

```

SQL> SELECT pkg.year_number FROM employees
2  WHERE employee_id = 138
3  /
SELECT pkg.year_number FROM employees
*
```

ERROR at line 1:

ORA-06553: PLS-221: 'YEAR_NUMBER' is not a procedure or is undefined

Классическое обходное решение основано на определении функции в пакете и ее последующем вызове:

```
SQL> CREATE OR REPLACE PACKAGE pkg
 2  IS
 3      FUNCTION year_number
 4          RETURN INTEGER;
 5  END;
 6  /
```

Package created.

```
SQL> CREATE OR REPLACE PACKAGE BODY pkg
 2  IS
 3      c_year_number  CONSTANT INTEGER := 2013;
 4
 5      FUNCTION year_number
 6          RETURN INTEGER
 7      IS
 8          BEGIN
 9              RETURN c_year_number;
10      END;
11  END;
12  /
```

Package body created.

```
SQL> SELECT pkg.year_number
 2  FROM employees
 3  WHERE employee_id = 138
 4  /
```

```
YEAR_NUMBER
-----
      2013
```

Для простого обращения к значению константы в команде SQL потребуется слишком много кода и усилий. Начиная с версии 12.1 это стало излишним — достаточно создать функцию в секции WITH:

```
WITH
  FUNCTION year_number
  RETURN INTEGER
  IS
  BEGIN
      RETURN pkg.year_number;
  END;
SELECT year_number
  FROM employees
 WHERE employee_id = 138
```

Функции PL/SQL, определяемые в SQL, также пригодятся при работе с автономными базами данных, доступными только для чтения. Хотя в таких базах данных невозможно создавать «вспомогательные» функции PL/SQL, вы можете определять их прямо в запросах.

Механизм WITH FUNCTION стал чрезвычайно полезным усовершенствованием языка SQL. Тем не менее каждый раз, когда вы планируете его использование, стоит задать себе один вопрос: «Потребуется ли эта функциональность в нескольких местах приложения?»

Если вы ответите на него положительно, следует решить, компенсирует ли выигрыш по производительности от применения WITH FUNCTION потенциальные потери от копирования и вставки этой логики в нескольких командах SQL.

Учтите, что в версии 12.1 в блоках PL/SQL невозможно выполнить статическую команду SELECT с секцией WITH FUNCTION. Безусловно, это выглядит очень странно, и я уверен, что в 12.2 такая возможность появится, но пока при попытке выполнения следующего кода будет выдана ошибка:

```
SQL> BEGIN
  2   WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
  3       RETURN VARCHAR2
  4       IS
  5       BEGIN
  6         RETURN fname_in || ' ' || lname_in;
  7       END;
  8
  9   SELECT LENGTH (full_name (first_name, last_name))
 10     INTO c
 11     FROM employees;
 12
 13   DBMS_OUTPUT.put_line ('count = ' || c);
 14 END;
 15 /
WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
      *
```

ERROR at line 2:
ORA-06550: line 2, column 18:
PL/SQL: ORA-00905: missing keyword



Помимо конструкции WITH FUNCTION, в версии 12.1 также появилась директива UDF для улучшения быстродействия функций PL/SQL, выполняемых из SQL. За подробностями обращайтесь к главе 21.

Табличные функции

Табличной функцией называется функция, которая может вызываться из секции FROM запроса, как если бы она была реляционной таблицей. Коллекции, возвращаемые табличными функциями, можно преобразовать оператором TABLE в структуру, к которой можно обращаться с запросами из языка SQL. Табличные функции особенно удобны в следующих ситуациях:

- Выполнение очень сложных преобразований данных, требующих использования PL/SQL, но с необходимостью обращаться к этим данным из команд SQL.
- Возвращение сложных результирующих наборов управляющей среде (отличной от PL/SQL). Вы можете открыть курсорную переменную для запроса, основанного на табличной функции, чтобы управляющая среда могла выполнить выборку данных через курсорную переменную.

Табличные функции открывают массу полезных возможностей для разработчиков PL/SQL. Чтобы продемонстрировать некоторые из этих возможностей, мы поближе познакомимся с потоковыми и конвейерными табличными функциями.

Потоковые табличные функции

Потоковая передача данных позволяет переходить между процессами или стадиями без использования вспомогательных структур данных. Табличные функции в сочетании с выражением CURSOR позволяют организовать потоковую передачу данных через несколько промежуточных преобразований в одной команде SQL.

Конвейерные табличные функции

Эти функции возвращают результирующий набор в *конвейерном* режиме, то есть данные поступают, пока функция продолжает выполняться. Добавьте секцию `PARALLEL_ENABLE` в заголовок конвейерной функции — и у вас появляется функция, которая будет выполняться параллельно в параллельном запросе.



До выхода Oracle Database 12c табличные функции могли возвращать только вложенные таблицы и `VARRAY`. Начиная с версии 12.1 появилась возможность определения табличных функций, возвращающих ассоциативные массивы с целочисленными индексами, тип которых определяется в спецификации пакета.

Давайте посмотрим, как определяются табличные функции и как использовать их в приложениях.

Вызов функции из секции FROM

Чтобы вызвать функцию из секции `FROM`, необходимо сделать следующее:

- Определить тип данных `RETURN` функции как тип коллекции (вложенная таблица или `VARRAY`).
- Убедиться в том, что все остальные параметры функции имеют режим `IN` и тип данных `SQL`. (Например, из запроса не удастся вызвать функцию, аргумент которой относится к логическому типу или типу записи.)
- Встроить вызов функции в оператор `TABLE` (в Oracle8i придется использовать оператор `CAST`).

Рассмотрим простой пример использования табличной функции. Мы начнем с создания типа вложенной таблицы на базе объектного типа `pets`:

```
/* Файл в Сети: pet_family.sql */
CREATE TYPE pet_t IS OBJECT (
    name  VARCHAR2 (60),
    breed  VARCHAR2 (100),
    dob    DATE);
CREATE TYPE pet_nt IS TABLE OF pet_t;
```

Затем создается функция с именем `pet_family`. В ее аргументах передаются два объекта `pet`. Далее в зависимости от значения `breed` возвращается вложенная таблица с информацией обо всем семействе, определенной в коллекции:

```
FUNCTION pet_family (dad_in IN pet_t, mom_in IN pet_t)
    RETURN pet_nt
IS
    l_count PLS_INTEGER;
    retval  pet_nt := pet_nt ();

    PROCEDURE extend_assign (pet_in IN pet_t) IS
    BEGIN
        retval.EXTEND;
        retval (retval.LAST) := pet_in;
    END;

    BEGIN
        extend_assign (dad_in);
        extend_assign (mom_in);

        IF    mom_in.breed = 'RABBIT' THEN l_count := 12;
        ELSIF mom_in.breed = 'DOG'    THEN l_count := 4;
        ELSIF mom_in.breed = 'KANGAROO' THEN l_count := 1;
        END IF;
```

```

FOR indx IN 1 .. l_count
LOOP
    extend_assign (pet_t ('BABY' || indx, mom_in.breed, SYSDATE));
END LOOP;

RETURN retval;
END;
```



Функция `pet_family` тривиальна; здесь важно понять, что функция PL/SQL может содержать сколь угодно сложную логику, которая реализуется средствами PL/SQL и выходит за рамки выразительных возможностей SQL.

Теперь эта функция может вызываться в секции `FROM` запроса:

```

SELECT pets.NAME, pets.dob
FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                        , pet_t ('Hippy', 'RABBIT', SYSDATE)
                        )
            ) pets;
```

Часть выходных данных:

| NAME | DOB |
|--------|-----------|
| ----- | ----- |
| Hoppy | 27-FEB-02 |
| Hippy | 27-FEB-02 |
| BABY1 | 27-FEB-02 |
| BABY2 | 27-FEB-02 |
| ... | |
| BABY11 | 27-FEB-02 |
| BABY12 | 27-FEB-02 |

Передача результатов вызова табличной функции в курсорной переменной

Табличные функции помогают решить проблему, с которой разработчики сталкивались в прошлом, — а именно как передать данные, полученные в программе PL/SQL (то есть данные, не хранящиеся в таблицах базы данных), в управляющую среду без поддержки PL/SQL? Курсорные переменные позволяют легко передать результирующие наборы на базе SQL, допустим, в программу Java, потому что курсорные переменные поддерживаются в JDBC. Но если сначала нужно провести сложные преобразования в PL/SQL, как вернуть эти данные вызывающей программе?

Теперь эта проблема решается объединением мощи и гибкости табличных функций с широкой поддержкой курсорных переменных в средах без поддержки PL/SQL (см. главу 15).

Допустим, я хочу сгенерировать данные семейства животных (полученные вызовом функции `pet_family` из предыдущего раздела) и передать строки данных интерфейсному приложению, написанному на Java. Это делается очень просто:

```

/* Файл в Сети: pet_family.sql */
FUNCTION pet_family_cv
RETURN SYS_REFCURSOR
IS
    retval SYS_REFCURSOR;
BEGIN
    OPEN retval FOR
        SELECT *
        FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                                , pet_t ('Hippy', 'RABBIT', SYSDATE)
                                )
                );
```

```
RETURN retval;
END pet_family_cv;
```

В этой программе я воспользуюсь преимуществами предопределенного слабого курсорного типа `SYS_REFCURSOR` (появившегося в Oracle9i Database) для объявления курсорной переменной. Курсорная переменная открывается вызовом `OPEN FOR` и связывается с запросом, построенным на базе табличной функции `pet_family`.

Затем курсорная переменная передается интерфейсной части Java. Так как JDBC распознает курсорные переменные, код Java легко выполняет выборку строк данных и интегрирует их в приложение.

Создание потоковой функции

Потоковая функция получает параметр с результирующим набором (через выражение `CURSOR`) и возвращает результат в форме коллекции. Так как к коллекции можно применить оператор `TABLE`, а затем запросить данные командой `SELECT`, эти функции позволяют выполнить одно или несколько преобразований данных в одной команде SQL.

Потоковые функции, поддержка которых добавилась в Oracle9i Database, позволяют скрыть алгоритмическую сложность за интерфейсом функции, и упростить SQL приложения. Приведенный ниже пример объясняет различные действия, которые необходимо выполнить для такого использования табличных функций.

Представьте следующую ситуацию: имеется таблица с информацией биржевых котировок, которая содержит строки с ценами на моменты открытия и закрытия биржи:

```
/* Файл в Сети: tabfunc_streaming.sql */
TABLE stocktable (
  ticker VARCHAR2(10),
  trade_date DATE,
  open_price NUMBER,
  close_price NUMBER)
```

Эту информацию необходимо преобразовать в другую таблицу:

```
TABLE tickertable (
  ticker VARCHAR2(10),
  pricetype DATE,
  pricetype VARCHAR2(1),
  price NUMBER)
```

Иначе говоря, одна строка `stocktable` превращается в две строки `tickertable`. Эту задачу можно решить многими способами. Самое элементарное и традиционное решение на PL/SQL выглядит примерно так:

```
FOR rec IN (SELECT * FROM stocktable)
LOOP
  INSERT INTO tickertable
    (ticker, pricetype, price)
  VALUES (rec.ticker, 'O', rec.open_price);

  INSERT INTO tickertable
    (ticker, pricetype, price)
  VALUES (rec.ticker, 'C', rec.close_price);
END LOOP;
```

Также возможны решения, полностью основанные на SQL:

```
INSERT ALL
  INTO tickertable
    (ticker, pricetype, price)
  VALUES (rec.ticker, 'O', rec.open_price),
  (rec.ticker, 'C', rec.close_price)
  FROM stocktable rec;
```

```
VALUES (ticker, trade_date, 'O', open_price
      )
      INTO tickertable
      (ticker, pricetype, price
      )
VALUES (ticker, trade_date, 'C', close_price
      )
SELECT ticker, trade_date, open_price, close_price
FROM stocktable;
```

А теперь предположим, что для перемещения данных из `stocktable` в `tickertable` требуется выполнить очень сложное преобразование, требующее использования PL/SQL. В такой ситуации табличная функция, используемая для передачи преобразуемых данных, потребует намного более эффективного решения.

Прежде всего, при использовании табличной функции нужно будет возвращать вложенную таблицу или массив `VARRAY` с данными. Я выбрал вложенную таблицу, потому что для `VARRAY` нужно задать максимальный размер, а я не хочу устанавливать это ограничение в своей реализации. Тип вложенной таблицы должен быть определен как тип на уровне схемы или в спецификации пакета, чтобы ядро SQL могло разрешить ссылку на коллекцию этого типа. Конечно, хотелось бы вернуть вложенную таблицу, основанную на самом определении таблицы, — то есть чтобы определение выглядело примерно так:

```
TYPE tickertype_nt IS TABLE OF tickertable%ROWTYPE;
```

К сожалению, эта команда завершится неудачей, потому что `%ROWTYPE` не относится к числу типов, распознаваемых SQL. Этот атрибут доступен только в разделе объявлений PL/SQL. Следовательно, вместо этого придется создать объектный тип, который воспроизводит структуру реляционной таблицы, а затем определить тип вложенной таблицы на базе этого объектного типа:

```
TYPE TickerType AS OBJECT (
  ticker VARCHAR2(10),
  pricetype DATE
  pricetype VARCHAR2(1),
  price NUMBER);
```

```
TYPE TickerTypeSet AS TABLE OF TickerType;
```

Чтобы табличная функция передавала данные с одной стадии преобразования на другую, она должна получать аргумент с набором данных — фактически запрос. Это можно сделать только одним способом — передачей курсорной переменной, поэтому в списке параметров функции необходимо будет использовать тип `REF CURSOR`.

Я создал пакет для типа `REF CURSOR`, основанного на новом типе вложенной таблицы:

```
PACKAGE refcur_pkg
IS
  TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
```

Работа завершается написанием функции преобразования:

```
/* Файл в Сети: tabfunc_streaming.sql */
1  FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
2    RETURN tickertypeset
3  IS
4    l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
5    l_row_from_query dataset%ROWTYPE;
6    retval tickertypeset := tickertypeset ();
7  BEGIN
8    LOOP
9      FETCH dataset
10     INTO l_row_from_query;
```



```

11
12     EXIT WHEN dataset%NOTFOUND;
13     -- Создание экземпляра объектного типа для начальной цены
14     l_row_as_object.ticker := l_row_from_query.ticker;
15     l_row_as_object.pricetype := 'O';
16     l_row_as_object.price := l_row_from_query.open_price;
17     l_row_as_object.pricedate := l_row_from_query.trade_date;
18     retval.EXTEND;
19     retval(retval.LAST) := l_row_as_object;
20     -- Создание экземпляра объектного типа для конечной цены
21     l_row_as_object.pricetype := 'C';
22     l_row_as_object.price := l_row_from_query.close_price;
23     retval.EXTEND;
24     retval(retval.LAST) := l_row_as_object;
25 END LOOP;
26
27 CLOSE dataset;
28
29 RETURN retval;
30 END stockpivot;

```

Как и в случае с функцией `pet_family`, конкретный код не важен; в ваших программах логика преобразований будет качественно сложнее. Впрочем, основная последовательность действий с большой вероятностью будет повторена в вашем коде, поэтому я приведу краткую сводку в следующей таблице.

| Строки | Описание |
|--------|---|
| 1–2 | Заголовок функции: получает результирующий набор в курсорной переменной, возвращает вложенный тип, основанный на объектном типе |
| 4 | Объявление локального объекта, который будет использоваться для заполнения вложенной таблицы |
| 5 | Объявление локальной записи, основанной на результирующем наборе. Будет заполняться вызовом <code>FETCH</code> для курсорной переменной |
| 6 | Локальная вложенная таблица, которая будет возвращаться функцией |
| 8–12 | Начало простого цикла с выборкой каждой строки из курсорной переменной; цикл завершается, когда в курсоре не остается данных |
| 14–19 | Использование «начальных» данных в записи для заполнения локального объекта и его включение во вложенную таблицу после определения новой строки (<code>EXTEND</code>) |
| 21–25 | Использование «конечных» данных в записи для заполнения локального объекта и его включение во вложенную таблицу после определения новой строки (<code>EXTEND</code>) |
| 27–30 | Закрытие курсора и возвращение вложенной таблицы |

Итак, теперь у меня имеется функция, которая будет проделывать всю нетривиальную, но необходимую работу, и я могу использовать ее в запросе для передачи данных между таблицами:

```

BEGIN
    INSERT INTO tickertable
    SELECT *
    FROM TABLE (stockpivot (CURSOR (SELECT *
                                     FROM stocktable)));
END;

```

Внутренняя команда `SELECT` извлекает все строки таблицы `stocktable`. Выражение `CURSOR`, в которое заключен запрос, преобразует итоговый набор в курсорную переменную, которая передается `stockpivot`. Функция возвращает вложенную таблицу, а оператор `TABLE` преобразует ее к формату реляционной таблицы, к которой можно обращаться с запросами.

Никакого волшебства, и все же выглядит немного волшебно, правда? Но вас ждет нечто еще более интересное — конвейерные функции!

Создание конвейерной функции

Конвейерной функцией называется табличная функция, которая возвращает результирующий набор как коллекцию, но делает это асинхронно с завершением самой функции. Другими словами, база данных уже не ожидает, пока функция отработает до конца и сохранит все вычисленные строки в коллекции PL/SQL, прежде чем выдать первые строки. Каждая запись, готовая к присваиванию в коллекцию, передается функцией как по конвейеру. В этом разделе описаны основы построения конвейерных табличных функций. О том, как эти функции влияют на производительность, подробно рассказано в главе 21. Чтобы лучше понять, что необходимо для построения конвейерных функций, мы переработаем функцию `stockpivot`:

```

/* Файл в Сети: tabfunc_pipelined.sql */
1  FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
2  RETURN tickertypeset PIPELINED
3  IS
4      l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
5      l_row_from_query dataset%ROWTYPE;
6  BEGIN
7      LOOP
8          FETCH dataset INTO l_row_from_query;
9          EXIT WHEN dataset%NOTFOUND;
10
11         -- первая строка
12         l_row_as_object.ticker := l_row_from_query.ticker;
13         l_row_as_object.pricetype := 'O';
14         l_row_as_object.price := l_row_from_query.open_price;
15         l_row_as_object.pricedate := l_row_from_query.trade_date;
16         PIPE ROW (l_row_as_object);
17
18         -- вторая строка
19         l_row_as_object.pricetype := 'C';
20         l_row_as_object.price := l_row_from_query.close_price;
21         PIPE ROW (l_row_as_object);
22     END LOOP;
23
24     CLOSE dataset;
25     RETURN;
26 END;
```

В следующей таблице перечислены некоторые изменения в исходной функциональности.

| Строки | Описание |
|------------------|--|
| 2 | По сравнению с исходной версией <code>stockpivot</code> добавлено ключевое слово <code>PIPELINED</code> |
| 4–5 | Объявление локального объекта и локальной записи, как и в первой версии. В этих строках интересно то, что <i>не</i> объявляется, — а именно вложенная таблица, которая будет возвращаться функцией. Намек на то, что будет дальше... |
| 7–9 | Начало простого цикла с выборкой каждой строки из курсорной переменной; цикл завершается, когда в курсоре не остается данных |
| 12–15 и 19–21 | Заполнение локального объекта для строк <code>tickertable</code> (на моменты открытия и закрытия) |
| 16 и 21 | Команда <code>PIPE ROW</code> (допустимая только в конвейерных функциях) немедленно передает объект, подготовленный функцией |
| 25 | В конце исполняемого раздела функция ничего не возвращает! Вместо этого она вызывает <code>RETURN</code> без указания значения (что прежде разрешалось только в процедурах) для возврата управления вызывающему блоку. Функция уже вернула все свои данные командами <code>PIPE ROW</code> |

Конвейерная функция вызывается так же, как и неконвейерная. Внешне никакие различия в поведении не проявляются (если только вы не настроили конвейерную функцию

для параллельного выполнения в составе параллельного запроса — см. следующий раздел — или не включили логику, использующую асинхронное возвращение данных). Возьмем запрос, использующий псевдостолбец `ROWNUM` для ограничения строк, включаемых в выборку:

```
BEGIN
  INSERT INTO tickertable
    SELECT *
      FROM TABLE (stockpivot (CURSOR (SELECT *
                                         FROM stocktable)))
    WHERE ROWNUM < 10;
END;
```

Мои тесты показывают, что в Oracle Database 10g и Oracle Database 11g при преобразовании 100 000 строк в 200 000 и последующем возвращении только первых 9 строк конвейерная версия завершает свою работу за 0,2 секунды, тогда как выполнение неконвейерной версии занимает 4,6 секунды.

Как видите, конвейерная передача строк работает, и обеспечивает существенный выигрыш!

Активизация параллельного выполнения функции

Одним из огромных достижений PL/SQL, появившихся в Oracle9i Database, стала возможность выполнения функций в контексте параллельных запросов. До выхода Oracle9i Database вызов функции PL/SQL в SQL переводил запрос в режим последовательного выполнения — существенная проблема для хранилищ данных большого объема. Теперь в заголовок конвейерной функции можно добавить информацию, которая подскажет исполнительному ядру, каким образом передаваемый функции набор данных следует разбить для параллельного выполнения.

В общем случае функция, предназначенная для параллельного выполнения, должна иметь один входной сильнотипизированный параметр `REF CURSOR`¹.

Несколько примеров:

- Функция может выполняться параллельно, а данные, передаваемые этой функции, могут разбиваться произвольно:

```
FUNCTION my_transform_fn (
  p_input_rows in employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
PARALLEL_ENABLE ( PARTITION p_input_rows BY ANY )
```

В этом примере ключевое слово `ANY` выражает утверждение программиста о том, что результаты не зависят от порядка получения входных строк функцией. При использовании этого ключевого слова исполнительная система случайным образом разбивает данные между процессами запроса. Это ключевое слово подходит для функций, которые получают одну строку, работают с ее столбцами, а затем генерируют выходные строки по содержимому столбцов только этой строки. Если в вашей программе действуют другие зависимости, результат становится непредсказуемым.

- Функция может выполняться параллельно, все строки заданного отдела должны передаваться одному процессу, а передача осуществляется последовательно:

```
FUNCTION my_transform_fn (
  p_input_rows in employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
```

продолжение ➤

¹ Входной параметр `REF CURSOR` может не иметь сильной типизации в режиме `ANY`.

```
CLUSTER P_INPUT_ROWS BY (department)
PARALLEL_ENABLE
( PARTITION P_INPUT_ROWS BY HASH (department) )
```

Oracle называет такой способ группировки записей *кластерным*; столбец, по которому осуществляется группировка (в данном случае `department`), называется *кластерным ключом*. Здесь важно то, что для алгоритма несущественно, в каком порядке значений кластерного ключа он будет получать кластеры, и Oracle не гарантирует никакого конкретного порядка получения. Тем самым обеспечивается ускорение работы алгоритма по сравнению с кластеризацией и передачей строк в порядке значений кластерного ключа. Алгоритм выполняется со сложностью N вместо $N \log(N)$, где N — количество записей.

В данном примере в зависимости от имеющейся информации о распределении значений можно выбрать между `HASH (department)` и `RANGE (department)`. `HASH` работает быстрее, и является более естественным вариантом для использования с `CLUSTER...BY`.

- Функция должна выполняться параллельно, а строки, передаваемые конкретному процессу в соответствии с `PARTITION...BY` (для этого раздела), будут проходить локальную сортировку этим процессом.

```
FUNCTION my_transform_fn (
    p_input_rows in employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
ORDER P_INPUT_ROWS BY (C1)
PARALLEL_ENABLE
( PARTITION P_INPUT_ROWS BY RANGE (C1) )
```

Фактически происходит параллелизация сортировки, поэтому команда `SELECT`, используемая для вызова табличной функции, не должна содержать секции `ORDER...BY` (так как ее присутствие будет противоречить попытке параллелизации сортировки). Следовательно, в данном случае естественно использовать вариант `RANGE` в сочетании с `ORDER...BY`. Реализация будет работать медленнее, чем `CLUSTER...BY`, поэтому этот вариант следует использовать только в том случае, если алгоритм зависит от него.



Конструкция `CLUSTER...BY` не должна использоваться вместе с `ORDER...BY` в объявлении табличной функции. Это означает, что алгоритм, зависящий от кластеризации по одному ключу `c1` с последующим упорядочением набора записей с заданным значением `c1`, скажем, по `c2`, должен проходить параллелизацию с использованием `ORDER...BY` в объявлении табличной функции.

Детерминированные функции

Функция называется *детерминированной*, если при одном наборе параметров `IN` и `IN OUT` она всегда возвращает одно и то же значение. Отличительной чертой детерминированных функций является отсутствие побочных эффектов: все изменения, вносимые программой, отражаются в списке параметров.

Пример детерминированной функции, которая представляет собой простую инкапсуляцию `SUBSTR`:

```
FUNCTION betwnstr (
    string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)
RETURN VARCHAR2 IS
BEGIN
    RETURN (SUBSTR (string_in, start_in, end_in - start_in + 1));
END betwnstr;
```

Если при вызове функции передаются, например, строка «abcdef» (`string_in`), число 3 (`start_in`) и 5 (`end_in`), то сколько бы раз вы ни вызывали функцию `betwnStr` с этим набором параметров, она всегда будет возвращать строку «cde». Тогда почему бы Oracle не сохранить результат, связанный с конкретным набором аргументов? Ведь при следующем вызове функции с теми же параметрами можно получить результат, не выполняя код функции! Чтобы добиться подобного эффекта, включите предложение `DETERMINISTIC` в заголовок функции:

```
FUNCTION betwnstr (  
    string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)  
    RETURN VARCHAR2 DETERMINISTIC
```

Решение об использовании сохраненной копии возвращаемого результата (если такая копия доступна) принимается оптимизатором запросов Oracle. Сохраненные копии могут браться из материализованного представления, функционального индекса или повторного вызова одной функции в команде SQL.



Функция должна быть объявлена с ключевым словом `DETERMINISTIC`, чтобы она могла вызываться в выражении функционального индекса или из запроса материализованного представления с пометкой `REFRESH FAST` или `ENABLE QUERY REWRITE`. Кроме того, кэширование входных данных и результатов детерминированной функции выполняется только тогда, когда функция вызывается в команде SQL.

Детерминированная функция может улучшить производительность команд SQL, вызывающих такие функции. За дополнительной информацией о детерминированных функциях как механизме кэширования обращайтесь к главе 21. В этой главе также описан новый механизм кэширования результатов, появившийся в Oracle11g (`RESULT_CACHE`).

В Oracle нет надежного способа проверки отсутствия побочных эффектов у функций, объявленных детерминированными. Поэтому при использовании таких функций вся ответственность за последствия возлагается на программиста. Детерминированная функция не должна основываться на переменных пакета и иметь право изменять результирующие значения в базе данных.

Эффект от использования детерминированных функций (и их ограничения) продемонстрирован в файле `deterministic.sql` на сайте книги.

Результаты неявных курсоров (Oracle Database 12c)

Знаете, чего я терпеть не могу? Когда разработчик утверждает, что Transact SQL лучше PL/SQL — и приводит некий аспект, который *действительно* лучше. Обычно я начинаю доказывать, что PL/SQL на голову выше Transact SQL... но в конце концов, у каждого языка есть свои сильные и слабые стороны, а в Transact SQL уже давно поддерживается возможность создания процедур, которые просто выводят стандарт результирующего набора на экран. В PL/SQL до недавнего времени приходилось писать запрос, перебирать результирующий набор и вызывать `DBMS_OUTPUT.PUT_LINE` для отображения результатов.

Однако теперь эта функциональность появилась и в PL/SQL. Это добавление ориентировано в первую очередь на разработчиков, переходящих с Transact SQL на PL/SQL (добро пожаловать!), а также предназначено для тестирования (стало намного проще написать короткую процедуру для вывода содержимого таблицы).

В Oracle данная возможность реализуется расширением функциональности пакета `DBMS_SQL`. Ура! Еще одна причина не отказываться от проверенного временем пакета (который во многом потерял актуальность из-за NDS).

Допустим, я хочу вывести фамилии всех работников заданного отдела. Для этого можно написать следующую процедуру:

```
/* Файл в Сети: 12c_return_result.sql */
CREATE OR REPLACE PROCEDURE show_emps (
    department_id_in IN employees.department_id%TYPE)
IS
    l_cursor    SYS_REFCURSOR;
BEGIN
    OPEN l_cursor FOR
        SELECT last_name
        FROM employees
        WHERE department_id = department_id_in
        ORDER BY last_name;

    DBMS_SQL.return_result (l_cursor);
END;
/
```

Если выполнить ее в SQL*Plus для отдела с идентификатором 20, получим:

```
BEGIN
    show_emps (20);
END;
/
PL/SQL procedure successfully completed.
```

ResultSet #1

```
LAST_NAME
-----
Fay
Hartstein
```

Вы также можете воспользоваться процедурой DBMS_SQL.GET_NEXT_RESULT для получения следующего результата, возвращаемого вызовом RETURN_RESULT, в программе PL/SQL (вместо того, чтобы передавать его во внешнюю среду).

Модульный подход — в жизнь!

По мере развития PL/SQL и средств Oracle усложняются и приложения, разрабатываемые с их помощью. Компании строят свой бизнес на приложениях PL/SQL, причем эти приложения используются годами — *и даже десятилетиями*. Откровенно говоря, без хорошего владения приемами модуляризации в PL/SQL вы вряд ли сможете реализовать крупный проект.

Эта книга дает представление о модульной основе, на которой вы будете строить свой код. Вам еще предстоит многое узнать — прежде всего изучить многочисленные пакеты, которые Oracle Corporation включает в поставку утилит и самой базы данных (например, пакет DBMS_RLS для реализации средств безопасности на уровне записей или пакет UTL_TCP для реализации функциональности, связанной с протоколом TCP).

Однако в основе всех этих технологий должна лежать твердая приверженность принципам модуляризации и повторного использования кода. Всеми силами избегайте избыточности и жесткого кодирования значений и формул. Развивайте в себе ревностное стремление к модульным конструкциям из «черных ящиков», которые легко адаптируются к разным приложениям.

Овладев методами модуляризации, вы заметите, что тратите больше времени на *проектирование*, а процесс отладки проходит быстрее. Ваши программы становятся более понятными и удобными в сопровождении, а код — более элегантным. Словом, беритесь за дело и внедряйте модульный подход!

18

Пакеты

Пакет представляет собой сгруппированный по определенным правилам именованный набор элементов кода PL/SQL. Он обеспечивает логическую структуру для организации программ и других элементов PL/SQL: курсоров, типов данных и переменных. Пакеты обладают очень важными функциональными возможностями, включая возможность сокрытия логики и данных, а также определения глобальных данных, существующих в течение сеанса.

Для чего нужны пакеты?

Пакеты — очень важная составная часть языка PL/SQL, краеугольный камень любого сложного проекта. Чтобы это понять, необходимо рассмотреть основные преимущества пакетов.

- **Упрощение сопровождения и расширения приложений.** По мере того как все большая часть кодовой базы перемещается в режим сопровождения, качество приложений PL/SQL определяется не только их производительностью, но и простотой сопровождения. С этой точки зрения пакеты играют исключительно важную роль, поскольку они обеспечивают инкапсуляцию кода (в частности, они позволяют скрыть команды SQL за интерфейсом процедур), дают возможность определять константы для литералов и «волшебных» чисел, и группировать логически связанные функции. Пакетный подход к проектированию и реализации сокращает количество потенциальных сбоев в приложениях.
- **Повышение производительности приложений.** Во многих ситуациях использование пакетов повышает производительность и эффективность работы приложений. Определение постоянных структур данных уровня пакета позволяет кэшировать статические значения из базы данных. Это дает возможность избежать повторных запросов, а следовательно, значительно ускорить получение результата. Кроме того, подсистема управления памятью Oracle оптимизирована для доступа к откомпилированному коду пакетов (за подробностями обращайтесь к главе 24).
- **Исправление недостатков приложений или встроенных элементов.** Некоторые из существующих программных компонентов Oracle имеют недостатки; в частности, не лучшим образом реализованы важнейшие функции встроенных пакетов `UTL_FILE` и `DBMS_OUTPUT`. Мириться с ними не обязательно; можно разработать собственный пакет на базе существующего, исправив как можно больше проблем. Например, сценарий `do.pkg`, описанный в главе 17, предоставляет замену для встроенной функции `DBMS_OUTPUT.PUT_LINE` с добавлением перегрузки для типа `XMLType`. Подобного

результата можно достичь и с помощью отдельных функций и процедур, но решение с пакетами более предпочтительно.

- **Снижение необходимости в перекомпиляции кода.** Пакет обычно состоит из двух элементов: спецификации и тела. Внешние программы (не определенные в пакете) могут вызывать только программы, перечисленные в спецификации. Изменение и перекомпиляция тела пакета не отражается на работе этих внешних программ. Снижение необходимости в перекомпиляции кода является важнейшим фактором администрирования больших объемов программного кода приложений.

Концепция пакетов очень проста. Единственная сложность заключается в том, чтобы научиться эффективно применять в приложениях их богатые возможности. В этой главе мы начнем с рассмотрения простого пакета; вы увидите, что основные преимущества пакетов проявляются даже в тривиальном коде. Затем будет рассмотрен специальный синтаксис, используемый при определении пакетов.



Прежде чем приступить к рассмотрению преимуществ пакетов и описанию синтаксиса их определения, необходимо сделать одно важное замечание. Всегда стройте приложение на основе пакетов; избегайте отдельных процедур и функций. Даже если вам сейчас кажется, что для реализации определенной возможности достаточно одной процедуры или функции, в будущем к ней почти наверняка добавятся еще несколько. Когда вы поймете, что их лучше объединить в пакет, придется искать все вызовы процедур и функций и добавлять к ним префикс с именем пакета. Используйте пакеты с самого начала, избавьте себя от будущих проблем!

Демонстрация возможностей пакетов

Пакет состоит из двух частей — *спецификации* и *тела*. Спецификация является обязательной частью и определяет, как разработчик может использовать пакет: какие программы можно вызывать, какие курсоры открывать и т. д. Тело пакета — необязательная, но почти всегда присутствующая часть; она содержит код перечисленных в спецификации программ (и возможно, курсоров), а также другие необходимые элементы кода.

Предположим, нам нужна программа для получения полного имени сотрудника, которое хранится в базе данных в виде двух отдельных элементов: фамилии и имени. На первый взгляд кажется, что задача решается просто:

```
PROCEDURE process_employee (
    employee_id_in IN employees.employee_id%TYPE)
IS
    l_fullname VARCHAR2(100);
BEGIN
    SELECT last_name || ',' || first_name
        INTO l_fullname
        FROM employees
        WHERE employee_id = employee_id_in;
    ...
END;
```

Однако этот вроде бы тривиальный код обладает рядом скрытых недостатков:

- Длина переменной `l_fullname` жестко закодирована. Поскольку полное имя — производное значение, которое строится конкатенацией содержимого двух столбцов, лучше так не делать. Если длина столбцов `last_name` и/или `first_name` будет увеличена, код процедуры придется изменять.
- Жестко закодировано правило составления полного имени. Чем это плохо? Тем, что если через какое-то время пользователь захочет получить полное имя в формате «ИМЯ ФАМИЛИЯ», вам придется производить замену во многих местах кода.

- Наконец, этот очень распространенный запрос может встречаться в нескольких местах приложения. Дублирование кода SQL затрудняет сопровождение приложения и его оптимизацию.

Приложения должны строиться таким образом, чтобы избежать жесткого кодирования подобных элементов. Определение типа данных для полного имени, представление, запрос к базе данных и т. п. должны кодироваться один раз в строго определенном месте и быть доступны из любой точки приложения. Таким местом и является пакет.

Рассмотрим следующую спецификацию пакета:

```
/* Файлы в Сети: fullname.pkg, fullname.tst */
1  PACKAGE employee_pkg
2  AS
3      SUBTYPE fullname_t IS VARCHAR2 (200);
4
5      FUNCTION fullname (
6          last_in  employees.last_name%TYPE,
7          first_in employees.first_name%TYPE)
8          RETURN fullname_t;
9
10     FUNCTION fullname (
11         employee_id_in IN employees.employee_id%TYPE)
12         RETURN fullname_t;
13 END employee_pkg;
```

Фактически здесь *перечисляются* различные элементы, которые должны использоваться разработчиками. Важнейшие элементы кода представлены в следующей таблице.

| Строки | Описание |
|--------|---|
| 3 | Объявление нового типа данных fullname_t. В этой версии его максимальная длина составляет 200 символов, но впоследствии ее будет легко изменить |
| 5–8 | Объявление функции fullname, которая строит полное имя по фамилии и имени. Обратите внимание: способ построения полного имени в спецификации пакета не указан |
| 10–13 | Объявление второй функции с тем же именем fullname; новая версия получает первичный ключ таблицы и возвращает соответствующее ему полное имя. Это типичный пример перегрузки, о которой говорилось в главе 17 |

Но прежде чем рассматривать реализацию пакета, давайте перепишем исходный блок кода таким образом, чтобы в нем использовались элементы пакета (обратите внимание на точечный синтаксис, аналогичный синтаксису *таблица.столбец*):

```
DECLARE
    l_name employee_pkg.fullname_t;
    employee_id_in employees.employee_id%TYPE := 1;
BEGIN
    l_name := employee_pkg.fullname (employee_id_in);
    ...
END;
```

Переменная l_name объявляется с новым типом данных, а для присваивания ей нужного значения вызывается соответствующая функция этого же пакета. Таким образом, формула построения полного имени и SQL-запрос вынесены из кода приложения в специальный «контейнер» для всей функциональности, относящейся к обработке данных о сотрудниках. Код стал проще и лаконичнее. Если потребуется изменить формулу построения полного имени или увеличить размер его типа данных, достаточно внести соответствующие изменения в спецификацию или тело пакета и перекомпилировать его код.

Реализация employee_pkg выглядит так:

```
1  PACKAGE BODY employee_pkg
2  AS
3      FUNCTION fullname (
```

продолжение ➤

```

4      last_in employee.last_name%TYPE,
5      first_in employee.first_name%TYPE
6  )
7      RETURN fullname_t
8  IS
9  BEGIN
10     RETURN last_in || ', ' || first_in;
11  END;
12
13  FUNCTION fullname (employee_id_in IN employee.employee_id%TYPE)
14     RETURN fullname_t
15  IS
16     retval    fullname_t;
17  BEGIN
18     SELECT fullname (last_name, first_name) INTO retval
19     FROM employee
20     WHERE employee_id = employee_id_in;
21
22     RETURN retval;
23  EXCEPTION
24     WHEN NO_DATA_FOUND THEN RETURN NULL;
25
26     WHEN TOO_MANY_ROWS THEN errpkg.record_and_stop;
27  END;
28  END employee_pkg;
```

В следующей таблице перечислены важные элементы этого кода.

| Строки | Описание |
|--------|---|
| 3–11 | Функция-«обертка» для определения формата полного имени «ФАМИЛИЯ, ИМЯ» |
| 13–27 | Типичный запрос на выборку одной строки, выполняемый с помощью неявного курсора |
| 18 | Вызов функции fullname, возвращающей комбинацию двух компонентов имени |

Если теперь пользователь потребует изменить формат представления полного имени, нам не придется искать по всему приложению все вхождения `|| ', ' ||` — для установки нового формата достаточно модифицировать в пакете `employee_pkg` функцию `fullname`.

Основные концепции пакетов

Прежде чем переходить к подробному изучению синтаксиса и структуры пакетов, следует изучить некоторые концепции пакетов:

- **Соккрытие информации.** Соккрытие информации о системе или приложении обычно преследует две цели. Во-первых, возможности человека по работе со сложными системами ограничены. Исследования показали, что у среднего «мозга» при запоминании даже семи (плюс/минус двух) элементов в группе возникают проблемы. Таким образом, пользователь (или разработчик) освобождается от необходимости вникать в ненужные подробности и может сосредоточиться на действительно важных аспектах. Во-вторых, соккрытие информации препятствует доступу к закрытым сведениям. Например, разработчик может вызвать в своем приложении готовую функцию для вычисления некоторого значения, но при этом формула вычислений может быть секретной. Кроме того, в случае изменения формулы все модификации будут вноситься только в одном месте.
- **Общие и приватные элементы.** Концепция общих и приватных элементов тесно связана с концепцией соккрытия информации. *Общедоступный* код определяется в спецификации пакета и доступен любой схеме, обладающей для этого пакета привилегией `EXECUTE`. *Приватный* код виден только в пределах пакета. Внешние программы, работающие с пакетом, не видят приватный код и не могут использовать его.

Приступая к разработке пакета, вы решаете, какие из его элементов будут общими, а какие — приватными. Кроме того, тело пакета можно скрыть от других схем/разработчиков. В таком случае пакет используется для сокрытия деталей реализации программ. Это особенно полезно для изоляции переменных компонентов приложения — фрагментов кода, зависящих от платформы, часто меняющихся структур данных и временных обходных решений.

На ранних стадиях развития программы в теле пакета также могут реализоваться в виде «заглушек» с минимальным объемом кода, необходимым для компиляции пакета. Этот прием позволяет сосредоточиться на интерфейсах программы и их взаимных связях.

- **Спецификация пакета.** Она содержит определения всех общедоступных элементов пакета, на которые можно ссылаться извне. Спецификация напоминает большой раздел объявлений; она не содержит блоков PL/SQL или исполняемого кода. Из хорошо спроектированной спецификации разработчик может получить всю необходимую для использования пакета информацию и ему никогда не придется заглядывать «за интерфейс» (то есть в тело пакета, содержащее реализацию его компонентов).
- **Тело пакета.** Здесь находится весь код, который необходим для реализации элементов, определенных в спецификации пакета. Тело может содержать отсутствующие в спецификации личные элементы, на которые нельзя ссылаться извне пакета, в частности объявления переменных и определения пакетных модулей. Кроме того, в теле пакета может находиться исполняемый (*инициализационный*) раздел, который выполняется только один раз для инициализации пакета.
- **Инициализация.** Концепция инициализации хорошо известна любому программисту, однако в контексте пакетов она имеет особое значение. В данном случае инициализируется не отдельная переменная, а весь пакет путем выполнения кода произвольной сложности. При этом Oracle следит за тем, чтобы пакет инициализировался только один раз за сеанс.
- **Постоянство в течение сеанса.** Концепция постоянства (или сохраняемости) тоже хорошо знакома программистам. Когда вы подключаетесь к Oracle и выполняете программу, присваивающую значение переменной уровня пакета (то есть переменной, объявленной в пакете вне содержащихся в нем программ), эта переменная сохраняет значение в течение всего сеанса, даже если выполнение присвоившей его программы завершается.

Также существует концепция *сеансового постоянства*. Если я подключаюсь к базе данных Oracle (создаю сеанс) и выполняю программу, которая присваивает значение пакетной переменной (то есть переменной, объявленной в спецификации или теле пакета, за пределами всех входящих в него программ), то эта переменная продолжает существовать на всем протяжении сеанса и сохраняет свое значение даже при завершении программы, выполнившей присваивание.

Именно пакеты обеспечивают поддержку структур данных с сеансовым постоянством в языке PL/SQL.

Графическое представление приватности

Различия между общедоступными и приватными элементами пакета дают разработчикам PL/SQL беспрецедентные средства управления структурами данных и программами. Грэди Буч предложил визуальное средство описания этих аспектов пакета (которое было вполне естественно названо диаграммой Буча).

Взгляните на рис. 18.1. Обратите внимание на надписи «Внутренняя часть» и «Внешняя часть». Первая часть содержит тело пакета (*внутренняя реализация пакета*), а вторая — все программы, написанные вами и не являющиеся частью пакета (*внешние программы*).

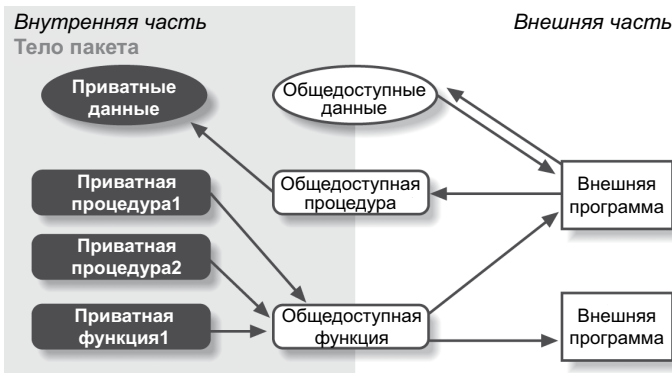


Рис. 18.1. Диаграмма Буча с общедоступными и приватными элементами пакета

Несколько выводов, следующих из диаграммы Буча:

- Внешние программы не могут пересекать границу внутренней реализации; иначе говоря, внешняя программа не может обращаться или вызывать элементы, определенные в теле пакета. Это приватные элементы, невидимые за пределами пакета.
- Элементы, определенные в спецификации пакета («Внешняя часть» на диаграмме), располагаются по обе стороны от границы между внутренней и внешней частью. Такие программы могут вызываться внешней программой (из внешней части), они доступны для частных программ и в свою очередь могут вызывать или обращаться ко всем остальным элементам пакета.
- Общедоступные элементы пакета предоставляют единственный путь к внутренней части. В этом отношении спецификация пакета действует как управляющий механизм для пакета в целом.
- Если окажется, что объект, ранее бывший приватным (например, модуль или курсор), должен стать общедоступным, просто добавьте его в спецификацию и перекомпилируйте пакет. После этого объект станет доступным за пределами пакета.

Правила построения пакетов

Структура пакета выглядит очень просто, но эта простота обманчива. Хотя синтаксис и правила построения пакетов вы изучите очень быстро, полное понимание нюансов реализации придет далеко не сразу. В этом разделе рассматриваются правила построения пакетов, а далее в этой главе рассказывается, в каких ситуациях пакеты особенно эффективны.

Чтобы создать пакет, необходимо написать его спецификацию и почти всегда — тело. При этом нужно решить, какие элементы пакета будут указаны в спецификации, а какие скрыты в теле. В пакет также можно включить блок кода, который Oracle будет выполнять при инициализации пакета.

Спецификация пакета

Спецификация пакета содержит список всех доступных элементов и предоставляет разработчику информацию, необходимую для использования пакета в приложениях. Ее часто называют программным интерфейсом — *API* (Application Programming Interface). Чтобы узнать, как применять описанные в спецификации элементы, разработчику не нужно изучать код, находящийся в теле пакета.

При разработке спецификации пакета необходимо руководствоваться следующими правилами:

- Элементы практически любого типа — числа, исключения, типы, коллекции и т. д. — могут объявляться на уровне пакета (то есть такие элементы не принадлежат конкретным процедурам или функциям этого пакета). Такие данные называются *данными уровня пакетов*. В общем случае объявлять переменные в спецификациях пакетов не рекомендуется, хотя объявления констант на уровне пакета вполне приемлемы. В пакете (как в спецификации, так и в теле) нельзя объявлять курсорные переменные (типа REF CURSOR), поскольку они не могут сохранять свое значение на протяжении сеанса (о постоянстве данных пакетов рассказано в разделе «Работа с данными пакета» далее в этой главе).
- В спецификации допускается объявление типов для любых структур данных: коллекций, записей или курсорных переменных.
- В спецификации можно объявлять процедуры и функции, но в ней должны быть указаны только их заголовки (часто определения процедуры или функции до ключевого слова IS или AS). Заголовок должен завершаться символом «;» (точка с запятой).
- В спецификацию пакета могут включаться явные курсоры. Они могут быть представлены в одной из двух форм: SQL-запрос либо является частью объявления курсора, либо скрывается в теле пакета (тогда в объявлении присутствует только предложение RETURN). Эта тема подробно рассматривается в разделе «Пакетные курсоры».
- Если в спецификации пакета объявляются процедуры или функции либо пакетный курсор без запроса, то тело пакета *должно* включать реализацию этих элементов.
- Спецификация пакета может содержать условие AUTHID, определяющее, как будут разрешаться ссылки на объекты данных: в соответствии с привилегиями владельца пакета (AUTHID DEFINER) или того, кто его вызывает (AUTHID CURRENT_USER). Подробнее об этом рассказывается в главе 24.
- После команды END в конце спецификации пакета можно разместить необязательную метку, идентифицирующую пакет:

END my_package;

Для демонстрации этих правил рассмотрим простую спецификацию пакета:

```
/* Файл в Сети: favorites.sql */
1  PACKAGE favorites_pkg
2    AUTHID CURRENT_USER
3  IS /* или AS */
4    -- Две константы: вместо малопонятных значений
5    -- используются информативные имена.
6
7    c_chocolate CONSTANT PLS_INTEGER := 16;
8    c_strawberry  CONSTANT PLS_INTEGER := 29;
9
10   -- Объявление типа вложенной таблицы
11   TYPE codes_nt IS TABLE OF INTEGER;
12
13   -- Вложенная таблица, объявленная на основе типа.
14   my_favorites codes_nt;
15
16   -- Курсорная переменная, возвращающая информацию из favorites.
17   TYPE fav_info_rct IS REF CURSOR RETURN favorites%ROWTYPE;
18
19   -- Процедура, принимающая список значений объявленного
20   -- выше типа codes_nt и выводящая соответствующую
21   -- информацию из таблицы..
22   PROCEDURE show_favorites (list_in IN codes_nt);
```

продолжение ⇨

```

23
24      -- Функция, возвращающая всю информацию из таблицы
25      -- favorites о самом популярном элементе.
26      FUNCTION most_popular RETURN fav_info_rct;
27
28  END favorites_pkg; -- Закрывающая метка пакета

```

Как видите, пакет имеет почти такую же структуру спецификации, как раздел объявлений блока PL/SQL. Единственное отличие заключается в том, что спецификация не может содержать кода реализации.

Тело пакета

Тело пакета содержит весь код, необходимый для реализации спецификации пакета. Оно не является стопроцентно необходимым; примеры спецификаций пакетов без тела приведены в разделе «Когда используются пакеты». Тело пакета необходимо в том случае, если истинны хотя бы некоторые из следующих условий:

- *Спецификация пакета содержит объявление курсора с секцией RETURN.* В этом случае команда SELECT должна быть указана в теле пакета.
- *Спецификация пакета содержит объявление процедуры или функции.* В этом случае реализация модуля должна быть завершена в теле пакета.
- *При инициализации пакета должен выполняться код, указанный в инициализационном разделе.* Спецификация пакета не поддерживает исполняемый раздел (исполняемые команды в блоке BEGIN-END); эти команды могут находиться только в теле пакета.

Со структурной точки зрения тело пакета очень похоже на определение процедуры. Несколько правил, специфических для тел пакетов:

- Тело пакета может содержать раздел объявлений, исполняемый раздел и раздел исключения. Раздел объявлений содержит полную реализацию всех курсоров и программ, определяемых в спецификации, а также определение всех приватных элементов (не указанных в спецификации). Раздел объявлений может быть пустым — при условии, что в теле пакета присутствует инициализационный раздел.
- Исполняемый раздел пакета также называется *инициализационным разделом*; он содержит дополнительный код, выполняемый при инициализации пакета в сеансе. Эта тема будет рассмотрена в следующем разделе.
- В разделе исключений обрабатываются все исключения, инициированные в инициализационном разделе. Раздел исключений может располагаться в конце тела пакета только в том случае, если вы определили инициализационный раздел.
- Тело пакета может иметь следующую структуру: только раздел объявлений; только исполняемый раздел; исполняемый раздел и раздел исключений; раздел объявлений, исполняемый раздел и раздел исключений.
- Секция AUTHID не может входить в тело пакета; она должна размещаться в спецификации пакета. Все, что объявлено в спецификации, может использоваться в теле пакета.
- Для тела и спецификации пакета действуют одни правила и ограничения объявления структур данных — например, невозможность объявления курсорных переменных.
- За командой END тела пакета может следовать необязательная метка с именем пакета:


```
END my_package;
```

Ниже приведена моя реализация тела favorites_pkg:

```

/* Файл в Сети: favorites.sql */
PACKAGE BODY favorites_pkg
IS
    -- Приватная переменная
    g_most_popular PLS_INTEGER := c_strawberry;

```

```

-- Реализация функции
FUNCTION most_popular RETURN fav_info_rct
IS
    retval fav_info_rct;
    null_cv fav_info_rct;
BEGIN
    OPEN retval FOR
        SELECT *
          FROM favorites
         WHERE code = g_most_popular;
    RETURN retval;
EXCEPTION
    WHEN NO_DATA_FOUND THEN RETURN null_cv;
END most_popular;

-- Реализация процедуры
PROCEDURE show_favorites (list_in IN codes_nt) IS
BEGIN
    FOR indx IN list_in.FIRST .. list_in.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE (list_in (indx));
    END LOOP;
END show_favorites;
END favorites_pkg; -- Метка конца пакета

```

Другие примеры тел пакетов приведены в разделе «Когда используются пакеты».

Инициализация пакетов

Пакет может содержать структуры данных, сохраняющиеся на протяжении всего сеанса (см. раздел «Работа с данными пакета»). Когда в ходе сеанса впервые происходит обращение к пакету (вызывается объявленная в нем программа, считывается или записывается значение переменной либо используется объявленный в пакете тип), Oracle инициализирует его, выполняя следующие действия:

- Создание экземпляров данных уровня пакетов (значения переменных и констант).
- Присваивание переменным и константам значений по умолчанию, указанных в объявлениях.
- Выполнение блока кода, содержащегося в *инициализационном разделе*.

Oracle выполняет все эти действия только один раз за сеанс и только тогда, когда возникнет непосредственная необходимость в этой информации.



Пакет может быть повторно инициализирован в ходе сеанса, если он был перекомпилирован с момента последнего использования или был выполнен сброс состояния всего сеанса, на что указывает ошибка ORA-04068.

Инициализационный раздел пакета составляют все операторы, находящиеся между ключевым словом **BEGIN** (вне определений процедур и функций) и ключевым словом **END**, завершающим тело пакета. Например, инициализационный раздел пакета **favorites_pkg** может выглядеть так:

```

/* Файл в Сети: favorites.sql */
PACKAGE BODY favorites_pkg
IS
    g_most_popular    PLS_INTEGER;

    PROCEDURE show_favorites (list_in IN codes_nt) ... END;

```

```
FUNCTION most_popular RETURN fav_info_rct ... END;

PROCEDURE analyze_favorites (year_in IN INTEGER) ... END;
-- Инициализационный раздел
BEGIN
  g_most_popular := c_chocolate;
  -- Функция EXTRACT используется для извлечения года из значения,
  -- возвращаемого функцией SYSDATE
  analyze_favorites (EXTRACT (YEAR FROM SYSDATE));
END favorites_pkg;
```

PL/SQL автоматически определяет, когда должен выполняться код инициализационного раздела. Это означает, что нет необходимости вызывать его явно и можно быть уверенными в том, что он будет выполнен только один раз. Когда следует использовать инициализационный раздел? Ниже описаны некоторые возможные причины.

Выполнение сложной логики инициализации

Конечно, значения по умолчанию могут присваиваться пакетным данным прямо в команде объявления. Тем не менее у этого подхода есть несколько потенциальных недостатков:

- Логика, необходимая для назначения значений по умолчанию, может быть слишком сложной для использования в конструкциях значений по умолчанию.
- Если при присваивании значения по умолчанию иницируется исключение, оно не может быть перехвачено в границах пакета; это исключение передается наружу необработанным. Эта тема более подробно рассматривается далее в разделе «Ошибки при инициализации».

Инициализация данных в инициализационном разделе обладает рядом преимуществ перед присваиванием значений по умолчанию. В частности, в исполняемом разделе вы обладаете полной гибкостью в определении, структуре и документировании ваших действий, а при возникновении исключения вы можете обработать его в разделе исключений инициализационного раздела.

Кэширование статической сеансовой информации

Другая причина для включения инициализационного раздела в пакет — кэширование *статической* информации, то есть остающейся неизменной на протяжении сеанса. Если значения данных не изменяются, зачем мириться с лишними затратами на запросы или повторное вычисление этих данных?

Кроме того, если вы хотите принять меры к тому, чтобы информация читалась в сеансе только один раз, инициализационный раздел становится идеальным автоматизированным решением.

При работе с кэшированными пакетными данными приходится учитывать важный компромисс между затратами памяти и вычислительных мощностей. Кэшируя данные в пакетных переменных, можно улучшить время выборки данных. Для этого данные размещаются «ближе» к пользователю, в области PGA каждого сеанса. При 1000 сеансах в системе существует 1000 копий кэшированных данных. Кэширование снижает нагрузку на процессор, но увеличивает затраты памяти — причем иногда весьма значительно.

За дополнительной информацией по этой теме обращайтесь к разделу «Кэширование статических данных сеанса для ускорения работы приложения».

Предотвращение побочных эффектов при инициализации

Избегайте присваивания значений глобальных данных в других пакетах (и вообще любых значений в других пакетах, если уж на то пошло). Эта защитная мера поможет

предотвратить хаос при выполнении кода и потенциальную путаницу у программистов, занимающихся сопровождением. Код инициализационного раздела должен быть сконцентрирован на текущем пакете. Помните, что этот код выполняется тогда, когда ваше приложение в первый раз пытается использовать элемент пакета. Пользователи не должны сидеть сложа руки, пока пакет выполняет высокзатратные вычисления, которые можно вынести в другие пакеты или триггеры приложения. Пример кода, которого следует избегать:

```
PACKAGE BODY company IS
BEGIN
    /*
    || Инициализационный раздел company_pkg обновляет глобальные
    || данные другого пакета. Ни в коем случае!
    */
    SELECT SUM (salary)
        INTO employee_pkg.max_salary
        FROM employees;
END company;
```

Если ваши требования к инициализации отличны от представленных нами, рассмотрите альтернативу для инициализационного раздела — например, сгруппируйте стартовые команды в процедуре приложения. Присвойте процедуре содержательное имя (например, `init_environment`); затем в нужной точке процесса инициализации вызовите процедуру `init_environment` для настройки сеанса.

Ошибки при инициализации

Инициализация пакета проходит в несколько этапов: объявление данных, присваивание значений по умолчанию, выполнение инициализационного раздела (если он присутствует). А если произойдет ошибка, приводящая к сбою процесса инициализации? Оказывается, даже если пакет не может завершить свои действия по инициализации, база данных помечает пакет как инициализированный и не пытается снова выполнять стартовый код в этом сеансе. Чтобы убедиться в этом, рассмотрим следующий пакет:

```
/* Файл в Сети: valerr.pkg */
PACKAGE valerr
IS
    FUNCTION get RETURN VARCHAR2;
END valerr;
PACKAGE BODY valerr
IS
    -- Глобальная - но при этом приватная - переменная уровня пакета
    v VARCHAR2(1) := 'ABC';
    FUNCTION get RETURN VARCHAR2
    IS
    BEGIN
        RETURN v;
    END;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Before I show you v...');
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE ('Trapped the error!');
END valerr;
```

Допустим, я подключаюсь к SQL*Plus и пытаюсь выполнить функцию `valerr.get` (первый раз в этом сеансе). Вот что я увижу:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE (valerr.get) *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

Иначе говоря, попытка объявления переменной `v` для присваивания значения «ABC» приводит к исключению `VALUE_ERROR`. Раздел исключений в конце пакета не перехватывает ошибку; он может перехватывать только те ошибки, которые иницируются в самом инициализационном разделе. Таким образом, исключение остается необработанным. Однако следует заметить, что при повторном вызове этой функции в сеансе ошибка уже не выдается:

```
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE ('V is set to ' || NVL (valerr.get, 'NULL'));
  3 END;
  4 /
V is set to NULL
```

Как интересно! Строка «Before I show you v...» вообще не выводится; более того, эта команда не выполняется. Ошибка происходит при первом вызове пакетной функции, но не при втором и всех последующих вызовах. Перед нами одна из классических «невоспроизводимых ошибок», а в мире PL/SQL это типичная причина подобных проблем: сбой в ходе инициализации пакета.

Подобные ошибки усложняют диагностику. Чтобы снизить риск таких ошибок и упростить их обнаружение, лучше всего переместить присваивание значений по умолчанию в инициализационный раздел, чтобы раздел исключений мог корректно обрабатывать ошибки и сообщать об их вероятных причинах:

```
PACKAGE BODY valerr
IS
  v VARCHAR2(1);
  FUNCTION get RETURN VARCHAR2 IS BEGIN ... END;
BEGIN
  v := 'ABC';
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE ('Error initializing valerr:');
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_STACK);
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END valerr;
```

Вы даже можете стандартизировать структуру пакетов и потребовать обязательного включения процедуры инициализации, чтобы разработчики группы не забывали об этой проблеме:

```
/* Файл в Сети: package_template.sql */
PACKAGE BODY <package_name>
IS
  -- Ниже размещаются приватные структуры данных.
  -- Не используйте присваивание значений по умолчанию.
  -- Вместо этого присваивайте их в процедуре инициализации
  -- и проверяйте успешность присваивания в программе проверки.
  -- Здесь размещаются приватные программы.

  -- инициализационный раздел (не обязателен)
  PROCEDURE initialize IS
  BEGIN
    NULL;
  END initialize;

  PROCEDURE verify_initialization (optional)
  -- Программа используется для проверки состояния пакета.
  -- Правильно ли прошло присваивание значений по умолчанию?
  -- Были ли выполнены все необходимые действия?
  IS
```

```

BEGIN
    NULL;
END verify_initialization;

-- Здесь размещаются открытые программы.

BEGIN
    initialize;
    verify_initialization;
END <имя_пакета>;
/

```

Правила вызова элементов пакета

Говорить о вызове или выполнении пакета в целом не имеет смысла, поскольку пакет является всего лишь контейнером для элементов кода. В программе можно использовать только элементы пакета.

Для ссылки на элементы пакета *извне* используется такой же точечный синтаксис, как для ссылки на столбцы таблиц. Давайте рассмотрим несколько примеров.

В следующей спецификации пакета объявляются константа, исключение, курсор и несколько модулей:

```

PACKAGE pets_inc
IS
    max_pets_in_facility CONSTANT INTEGER := 120;
    pet_is_sick EXCEPTION;

    CURSOR pet_cur (pet_id_in IN pet.id%TYPE) RETURN pet%ROWTYPE;

    FUNCTION next_pet_shots (pet_id_in IN pet.id%TYPE) RETURN DATE;
    PROCEDURE set_schedule (pet_id_in IN pet.id%TYPE);

END pets_inc;

```

Для ссылки на любые из этих элементов перед именем элемента ставится префикс в виде имени пакета:

```

DECLARE
    -- Константа объявляется на основе типа столбца id таблицы pet
    c_pet CONSTANT pet.id%TYPE:= 1099;
    v_next_appointment DATE;
BEGIN
    IF pets_inc.max_pets_in_facility > 100
    THEN
        OPEN pets_inc.pet_cur (c_pet);
    ELSE
        v_next_appointment:= pets_inc.next_pet_shots (c_pet);
    END IF;
EXCEPTION
    WHEN pets_inc.pet_is_sick
    THEN
        pets_inc.set_schedule (c_pet);
END;

```

Итак, при обращении к элементам пакета необходимо соблюдать два правила:

- Если элемент определяется в спецификации пакета, то для ссылки на него из внешней программы следует использовать точечный синтаксис: *имя_пакета.имя_элемента*.
- Если ссылка на элемент используется в самом пакете (спецификации или теле), имя пакета задавать не нужно, поскольку PL/SQL автоматически идентифицирует ее как обращение к элементу, объявленному в области действия пакета.

Работа с данными пакета

Данные пакета состоят из переменных и констант, определенных *на уровне пакета*, а не в конкретной его функции или процедуре. Их областью видимости является не отдельная программа, а весь пакет. Структуры данных пакета существуют (и сохраняют свои значения) на протяжении всего сеанса, а не только во время выполнения программы.

Если данные пакета объявлены в его теле, они сохраняются в течение сеанса, но доступны только элементам пакета (то есть являются приватными).

Если данные пакета объявлены в его спецификации, они также сохраняются в течение всего сеанса, но их чтение и изменение разрешено любой пользовательской программе, обладающей привилегией `EXECUTE` для пакета. Общие данные пакета похожи на глобальные переменные Oracle Forms (а их использование сопряжено с таким же риском).

Если пакетная процедура открывает курсор, он остается открытым и доступным в ходе всего сеанса. Нет необходимости объявлять курсор в каждой программе. Один модуль может его открыть, а другой — выполнить выборку данных. Переменные пакета могут использоваться для передачи данных между транзакциями, поскольку они привязаны не к транзакции, а к сеансу.

Глобальные данные в сеансе Oracle

В среде PL/SQL структуры данных пакета функционируют как глобальные. Однако следует помнить, что они доступны только в пределах одного сеанса или подключения к базе данных Oracle и не могут совместно использоваться несколькими сеансами. Если доступ к данным нужно обеспечить для нескольких сеансов Oracle, используйте пакет `DBMS_PIPE` (его описание имеется в документации *Oracle Built-In Packages*).

Будьте осторожны с предположением о том, что разные части приложения всегда работают с Oracle через одно подключение. В некоторых случаях среда, из которой выполняется компонент приложения, устанавливает для него новое подключение. При этом данные пакета, записанные первым подключением, будут недоступны для второго.

Допустим, приложение Oracle Forms сохранило значение в пакетной структуре данных. Когда форма вызывает хранимую процедуру, эта процедура может обращаться к тем же пакетным переменным и значениям, что и форма, потому что они используют одно подключение к базе данных. Но допустим, форма генерирует отчет с использованием Oracle Reports. По умолчанию Oracle Reports создает для отчета отдельное подключение к базе данных с тем же именем пользователя и паролем. Даже если отчет обратится к тому же пакету и структурам данных, что и форма, значения, хранимые в структурах данных, доступных форме и отчету, будут разными, поскольку сеанс отчета имеет свой экземпляр пакета и всех его структур.

По аналогии с двумя типами структур данных в пакетах (общедоступные и приватные) также существуют два типа глобальных данных пакетов: глобальные общедоступные данные и глобальные приватные данные. В следующих трех разделах рассматриваются различные способы использования данных пакетов.

Глобальные общедоступные данные

Любая структура данных, объявленная в спецификации пакета, является глобальной общедоступной структурой данных; это означает, что к ней может обратиться любая

программа за пределами пакета. Например, вы можете определить коллекцию PL/SQL в спецификации пакета и использовать ее для ведения списка работников, заслуживших повышение. Вы также можете создать пакет с константами, которые должны использоваться во всех программах. Другие разработчики будут ссылаться на пакетные константы вместо использования фиксированных значений в программах. Глобальные общедоступные структуры данных также могут изменяться, если только они не были объявлены с ключевым словом `CONSTANT`.

Глобальные данные обычно считаются источником повышенной опасности в программировании. Их очень удобно объявлять, они прекрасно подходят для того, чтобы вся информация была доступна в любой момент времени — однако зависимость от глобальных структур данных приводит к созданию неструктурированного кода с множеством побочных эффектов.

Вспомните, что спецификация модуля должна сообщать полную информацию, необходимую для вызова и использования этого модуля. Однако по спецификации пакета невозможно определить, выполняет ли пакет чтение и/или запись в глобальные структуры данных. По этой причине вы не можете быть уверены в том, что происходит в приложении и какая программа изменяет те или иные данные.

Передачу данных модулям и из них всегда рекомендуется осуществлять через параметры. В этом случае зависимость от структур данных документируется в спецификации и может учитываться разработчиками. С другой стороны, именованные глобальные структуры данных должны создаваться для информации, действительно глобальной по отношению к приложению — например, констант и параметров конфигурации.

Такие данные следует разместить в централизованном пакете. Однако учтите, что при такой архитектуре в приложении возникает «единая точка перекомпиляции»: каждый раз, когда вы вносите изменение в пакет и перекомпилируете спецификацию, многие программы приложения теряют работоспособность.

Пакетные курсоры

Одним из самых интересных типов пакетных данных является явный курсор (см. главу 14). Его можно объявить в теле либо в спецификации пакета. Состояние курсора (открыт или закрыт), а также указатель на его набор данных сохраняются в течение всего сеанса. Это означает, что открыть пакетный курсор можно в одной программе, выбрать из него данные — в другой, а закрыть — в третьей. Такая гибкость курсоров предоставляет большие возможности, но в то же время она может стать источником проблем.

Сначала мы рассмотрим некоторые тонкости объявления пакетных курсоров, а затем перейдем к открытию, выборке данных и закрытию таких курсоров.

Объявление пакетных курсоров

Явный курсор в спецификации пакета можно объявлять двумя способами:

- Полностью (заголовок курсора и запрос). Именно так объявляются курсоры в локальных блоках PL/SQL.
- Частично (только заголовок курсора). В этом случае запрос определяется в теле пакета, поэтому реализация курсора скрыта от использующего пакет разработчика.

При использовании второго способа в объявление нужно добавить секцию `RETURN`, указывающую, какие данные будут возвращены при выборке из курсора. На самом деле эти данные определяются инструкцией `SELECT`, которая присутствует только в теле, но не в спецификации.

В секции RETURN можно задать одну из следующих структур данных:

- запись, объявленная на основе таблицы базы данных с использованием атрибута %ROWTYPE;
- запись, определенная программистом.

Объявление курсора в теле пакета осуществляется так же, как в локальном блоке PL/SQL. Следующий пример спецификации пакета демонстрирует оба подхода:

```

/* Файл в Сети: pkgcur.sql */
1  PACKAGE book_info
2  IS
3      CURSOR byauthor_cur (
4          author_in IN books.author%TYPE
5      )
6  IS
7      SELECT *
8      FROM books
9      WHERE author = author_in;
10
11     CURSOR bytitle_cur (
12         title_filter_in IN books.title%TYPE
13     ) RETURN books%ROWTYPE;
14
15     TYPE author_summary_rt IS RECORD (
16         author books.author%TYPE,
17         total_page_count PLS_INTEGER,
18         total_book_count PLS_INTEGER);
19
20     CURSOR summary_cur (
21         author_in IN books.author%TYPE
22     ) RETURN author_summary_rt;
23 END book_info;

```

Логика программы описана в следующей таблице.

| Строки | Описание |
|--------|---|
| 3–9 | Типичное определение явного курсора, полностью заданное в спецификации пакета |
| 11–13 | Определение курсора без запроса. Спецификация указывает, что открыв курсор и выбрав из него данные, пользователь получит одну строку из таблицы books под действием заданного фильтра |
| 15–18 | Определение нового типа записи для хранения информации об авторе |
| 20–22 | Объявление курсора, возвращающего сводную информацию о заданном авторе (всего три значения) |

Рассмотрим тело пакета и выясним, какой код необходимо написать для работы с каждым из этих курсоров:

```

1  PACKAGE BODY book_info
2  IS
3      CURSOR bytitle_cur (
4          title_filter_in IN books.title%TYPE
5      ) RETURN books%ROWTYPE
6  IS
7      SELECT *
8      FROM books
9      WHERE title LIKE UPPER (title_filter_in);
10
11     CURSOR summary_cur (
12         author_in IN books.author%TYPE
13     ) RETURN author_summary_rt
14  IS
15     SELECT author, SUM (page_count), COUNT (*)
16     FROM books
17     WHERE author = author_in;
18 END book_info;

```

Поскольку у нас имеются два курсора с секциями `RETURN`, их определения нужно завершить в теле пакета. Список выбираемых запросом элементов в теле пакета должен соответствовать (по количеству элементов и их типам данных) секции `RETURN` в спецификации пакета. Если это условие не соблюдается, то при компиляции пакета вы получите одно из следующих сообщений об ошибке:

```
PLS-00323: subprogram or cursor '<cursor>' is declared in a
package specification and must be defined in the package body
PLS-00400: different number of columns between cursor SELECT
statement and return value
```

Работа с пакетными курсорами

Теперь давайте посмотрим, как пользоваться пакетными курсорами. Прежде всего для открытия, выборки данных и закрытия вам не придется изучать новый синтаксис — нужно только задать имя пакета перед именем курсора. Например, чтобы запросить информацию о книгах по PL/SQL, можно выполнить такой блок кода:

```
DECLARE
    onebook    book_info.bytitle_cur%ROWTYPE;
BEGIN
    OPEN book_info.bytitle_cur ('%PL/SQL%');
    LOOP
        EXIT WHEN book_info.bytitle_cur%NOTFOUND;
        FETCH book_info.bytitle_cur INTO onebook;
        book_info.display (onebook);
    END LOOP;

    CLOSE book_info.bytitle_cur;
END;
```

Как видите, на основе пакетного курсора точно так же можно объявить переменную с использованием `%ROWTYPE` и проверить атрибуты. Ничего нового!

Однако и в этом простом фрагменте кода есть скрытый нюанс. Поскольку курсор объявлен в спецификации пакета, его область видимости не ограничивается конкретным блоком PL/SQL. Предположим, мы выполняем следующий код:

```
BEGIN -- Только открываем курсор...
    OPEN book_info.bytitle_cur ('%PEACE%');
END;
```

Если затем в том же сеансе выполнить приведенный анонимный блок в приведенном выше цикле `LOOP`, то Oracle выдаст сообщение об ошибке (ORA-06511).

Дело в том, что блок, выполненный первым, не закрыл курсор, и по завершении его работы курсор остался открытым.

При работе с пакетными курсорами необходимо всегда соблюдать следующие правила:

- Никогда не рассчитывайте на то, что курсор закрыт (и готов к открытию).
- Никогда не рассчитывайте на то, что курсор открыт (и готов к закрытию).
- Всегда явно закрывайте курсор после завершения работы с ним. Эту логику также необходимо включить в обработчики исключений; убедитесь в том, что курсор закрывается на всех путях выхода из программы.

Если пренебречь этими правилами, то ваши приложения будут работать нестабильно, а в процессе их функционирования могут появиться неожиданные необработанные исключения. Поэтому лучше написать процедуры, которые открывают и закрывают курсоры и учитывают все возможные их состояния. Этот подход реализован в следующем пакете:

```
/* Файл в Сети: openclose.sql */
PACKAGE personnel
```

```

IS
  CURSOR emps_for_dept (
    department_id_in IN employees.department_id%TYPE)
  IS
    SELECT * FROM employees
      WHERE department_id = department_id_in;

  PROCEDURE open_emps_for_dept(
    department_id_in IN employees.department_id%TYPE,
    close_if_open IN BOOLEAN := TRUE
  );

  PROCEDURE close_emps_for_dept;

END personnel;
```

Как видите, вместе с курсором объявлены две сопутствующие процедуры для его открытия и закрытия. Если нам, скажем, потребуется перебрать в цикле строки курсора, это можно сделать так:

```

DECLARE
  one_emp personnel.emps_for_dept%ROWTYPE;
BEGIN
  personnel.open_emps_for_dept (1055);

  LOOP
    EXIT WHEN personnel.emps_for_dept%NOTFOUND;
    FETCH personnel.emps_for_dept INTO one_emp;
    ...
  END LOOP;

  personnel.close_emps_for_dept;
END;
```

В этом фрагменте не используются явные вызовы `OPEN` и `CLOSE`. Вместо них вызываются соответствующие процедуры, скрывающие особенности работы с пакетными курсорами. Реализация этих процедур представлена в файле `openclose.sql` на сайте книги.



Один из технических рецензентов этой книги Джей-Ти Томас предлагает взглянуть на ситуацию под другим углом:

«...Вместо того чтобы работать с пакетными курсорами, можно добиться точно такого же эффекта посредством инкапсуляции логики и данных в представлениях и опубликовать их для разработчиков. В этом случае разработчики будут нести ответственность за сопровождение своих курсоров; дело в том, что обеспечить нормальное сопровождение с инструментарием, существующим для общедоступных пакетных курсоров, невозможно. А именно, насколько мне известно, невозможно гарантировать использование процедур открытия/закрытия, но курсор всегда будет оставаться видимым для разработчика, который открывает/закрывает его напрямую; следовательно, такая конструкция остается уязвимой. Проблема усугубляется тем, что использование общедоступных пакетных курсоров и процедур открытия/закрытия может породить в группе ложное чувство безопасности и защищенности».

Пакетное создание курсоров и предоставление доступа к ним всем разработчикам, участвующим в проекте, приносит большую пользу. Проектирование оптимальных структур данных приложения — непростая и кропотливая работа. Эти структуры — и хранящиеся в них данные — используются в программах PL/SQL, а работа с ними почти всегда осуществляется через курсоры. Если вы не определите свои курсоры в пакетах и не предоставите их «в готовом виде» всем разработчикам, то каждый будет писать собственную реализацию курсора, а это создаст массу проблем с производительностью

и сопровождением кода. Пакетные курсоры являются лишь одним из примеров инкапсуляции доступа к структурам данных (подробнее см. далее в разделе «Когда используются пакеты»).

Повторно инициализируемые пакеты

По умолчанию пакетные данные сохраняются в течение всего сеанса (или до перекомпиляции пакета). Это исключительно удобное свойство пакетов, но и у него имеются определенные недостатки:

- Постоянство глобально доступных (общих и приватных) структур данных сопровождается нежелательными побочными эффектами. В частности, можно случайно оставить пакетный курсор открытым, а в другой программе попытаться открыть его без предварительной проверки, что приведет к ошибке.
- Если данные хранятся в структурах уровня пакетов, то программа может занять слишком большой объем памяти, не освобождая ее.

Для оптимизации использования памяти при работе с пакетами можно использовать директиву `SERIALLY_REUSABLE`. Она указывает Oracle, что пакет является *повторно инициализируемым*, то есть его состояние (состояние переменных, открытых пакетных курсоров и т. п.) нужно сохранять не на протяжении сеанса, а на время одного вызова пакетной программы.

Рассмотрим действие этой директивы на примере пакета `book_info`. В нем имеются две отдельные программы: для заполнения списка книг и для вывода этого списка.

```
/* Файл в Сети: serialpkg.sql */
PACKAGE book_info
IS
    PRAGMA SERIALLY_REUSABLE;
    PROCEDURE fill_list;

    PROCEDURE show_list;
END;
```

Как видно из приведенного ниже тела пакета, список объявляется как приватный глобальный ассоциативный массив:

```
/* Файл в Сети: serialpkg.sql */
PACKAGE BODY book_info
IS
    PRAGMA SERIALLY_REUSABLE;

    TYPE book_list_t
    IS
        TABLE OF books%ROWTYPE
            INDEX BY PLS_INTEGER;
    my_books book_list_t;

    PROCEDURE fill_list
    IS
    BEGIN
        FOR rec IN (SELECT *
                     FROM books
                     WHERE author LIKE '%FEUERSTEIN%')
        LOOP
            my_books (my_books.COUNT + 1) := rec;
        END LOOP;
    END fill_list;

    PROCEDURE show_list
```

продолжение ➤

```

IS
BEGIN
    IF my_books.COUNT = 0
    THEN
        DBMS_OUTPUT.PUT_LINE ('** Книг нет...');
    ELSE
        FOR indx IN 1 .. my_books.COUNT
        LOOP
            DBMS_OUTPUT.PUT_LINE (my_books (indx).title);
        END LOOP;
    END IF;
END show_list;
END;
```

Чтобы увидеть, как работает эта директива, заполним список и выведем его на экран. В первом варианте оба шага выполняются в одном блоке:

```

SQL> BEGIN
2     DBMS_OUTPUT.PUT_LINE (
3         'Заполнение и вывод в одном блоке:'
4     );
5     book_info.fill_list;
6     book_info.show_list;
7 END;
8 /
```

Заполнение и вывод в одном блоке:

```

Oracle PL/SQL Programming
Oracle PL/SQL Best Practices
Oracle PL/SQL Built-in Packages
```

Во второй версии заполнение и вывод списка производятся в разных блоках. В результате коллекция окажется пустой:

```

SQL> BEGIN
2     DBMS_OUTPUT.PUT_LINE ('Заполнение в первом блоке');
3     book_info.fill_list;
4 END;
5 /
```

Заполнение в первом блоке

```

SQL> BEGIN
2     DBMS_OUTPUT.PUT_LINE ('Вывод во втором блоке:');
3     book_info.show_list;
4 END;
5 /
```

Вывод во втором блоке:
 ** Нет книг...

Работая с повторно инициализируемыми пакетами, необходимо учитывать некоторые особенности:

- Глобальная память для такого пакета выделяется в системной глобальной области (SGA), а не в пользовательской глобальной области (UGA), что позволяет повторно применять рабочую область пакета. При каждом новом обращении к пакету его общие переменные инициализируются значениями по умолчанию или значением NULL, а его инициализационный раздел выполняется повторно.
- Максимальное количество рабочих областей, необходимых для повторно инициализируемого пакета, равно количеству одновременно работающих с этим пакетом пользователей. Увеличение объема используемой памяти в SGA компенсируется

уменьшением объема памяти, задействованного в UGA. Когда нужно предоставить память из SGA другим запросам, Oracle освобождает неиспользуемые области памяти.

Когда используются пакеты

Итак, мы рассмотрели синтаксис, основные правила и нюансы построения пакетов. Теперь можно вернуться к списку ситуаций, в которых пакеты следует применять, и рассмотреть их более подробно.

- **Инкапсуляция (сокрытие) операций с данными.** Вместо того чтобы заставлять разработчиков самих писать команды SQL (что усложняет сопровождение и снижает эффективность кода), предоставьте программный интерфейс (API) к этим командам SQL. Такой интерфейс называется *табличным*, или *транзакционным*.
- **Исключение жесткого кодирования литералов.** Чтобы избежать жесткого кодирования одних и тех же значений в нескольких программах, нужно определить все литеральные значения как константы и объединить их в пакеты. Можно также объявить константы в процедурах и функциях, но преимущество пакетных констант заключается в их общедоступности.
- **Устранение недостатков встроенных функций.** Некоторые встроенные средства Oracle (например, `UTL_FILE` и `DBMS_OUTPUT`) далеки от совершенства. На их базе можно создать собственный пакет, в котором значительная часть недостатков исправлена.
- **Группировка логически связанных функций.** Если у вас имеется десяток процедур и функций, относящихся к реализации некоторой части приложения, сгруппируйте их в пакет — это упростит управление и сопровождение кода.
- **Кэширование статических данных сеанса для ускорения работы приложения.** Для кэширования данных, не изменяющихся в течение сеанса, пользуйтесь постоянными структурами данных пакетов.

Ниже каждая из этих причин рассматривается более подробно.

Инкапсуляция доступа к данным

Вместо того чтобы заставлять разработчиков самостоятельно писать команды SQL, предоставьте интерфейс к этим командам. Это одна из самых важных причин для построения пакетов, но она применяется относительно редко.

При таком подходе разработчики PL/SQL вместо команд SQL обычно включают в свои приложения заранее определенный, протестированный и оптимизированный код, который выполняет всю работу за них; например, процедуру `add` (перегруженную для поддержки записей), которая выдает команду `INSERT` и следует стандартным правилам обработки ошибок, функцию для выборки одной записи по первичному ключу и разнообразные курсоры для обработки стандартных запросов к структуре данных (которой может быть одна таблица или «бизнес-сущность» из нескольких таблиц).

Если вы выберете этот путь, разработчикам не нужно будет разбираться, как объединить три или шесть нормализованных таблиц для получения нужного набора данных. Они просто выбирают нужный курсор, а анализ данных оставляется кому-то другому. Им не нужно думать, что делать, если при попытке вставки строки уже существует — процедура уже содержит всю необходимую логику.

Вероятно, самое серьезное преимущество такого подхода заключается в том, что при изменении структуры данных проблемы с обновлением кода приложения сводятся к минимуму и централизуются. Человек, хорошо знающий таблицу или объектный тип, вносит необходимые изменения в одном пакете, а эти изменения затем более или менее автоматически распространяются на всех программы, зависящие от этого пакета.

Инкапсуляция данных — весьма обширная и непростая тема. Пример пакета инкапсуляции таблицы (для таблицы `employees`) содержится в файлах `employee_tp.pks`, `employee_qp.*`, `employee_cp.*`, `department_tp.pks` и `department_qp.*` на сайте книги (эти файлы были сгенерированы программой Quest CodeGen Utility, которую можно загрузить на сайте ToadWorld).

Давайте посмотрим, как использование пакетов отражается на коде. Файл `givebonus1.sp` на сайте книги содержит процедуру, которая начисляет одинаковые премии всем работникам заданного отдела, — но только при условии, что стаж работника в компании составляет не менее 6 месяцев. Ниже приведены части программы `give_bonus` с кодом SQL (полная реализация содержится в файле `givebonus1.sp`):

```
/* Файл в Сети: givebonus1.sp */
PROCEDURE give_bonus (
    dept_in IN employees.department_id%TYPE,
    bonus_in IN NUMBER)
/*
|| Начисление премии каждому работнику заданного отдела,
|| но только при условии, что он проработал в компании
|| не менее 6 месяцев.
*/
IS
    l_name VARCHAR2(50);
    CURSOR by_dept_cur
    IS
        SELECT * FROM employees
        WHERE department_id = dept_in;

    fdbk INTEGER;
BEGIN
    /* Выборка всей информации по заданному отделу. */
    SELECT department_name INTO l_name
    FROM departments
    WHERE department_id = dept_in;

    /* Проверка идентификатора отдела. */
    IF l_name IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE (
            'Invalid department ID specified: ' || dept_in);
    ELSE
        /* Вывод заголовка. */
        DBMS_OUTPUT.PUT_LINE (
            'Applying Bonuses of ' || bonus_in ||
            ' to the ' || l_name || ' Department');
    END IF;
    /* Для каждого работника заданного отдела... */
    FOR rec IN by_dept_cur
    LOOP
        IF employee_rp.eligible_for_bonus (rec)
        THEN
            /* Обновление столбца. */

            UPDATE employees
            SET salary = rec.salary + bonus_in
            WHERE employee_id = rec.employee_id;
            END IF;
        END LOOP;
    END;
```

Сравните с альтернативным решением, полный код которого содержится в файле `givebonus2.sp`:

```

/* Файл в Сети: givebonus2.sp */
1  PROCEDURE give_bonus (
2    dept_in   IN   employee_tp.department_id_t
3    , bonus_in IN   employee_tp.bonus_t
4  )
5  IS
6    l_department    department_tp.department_rt;
7    l_employees     employee_tp.employee_tc;
8    l_rows_updated  PLS_INTEGER;
9  BEGIN
10   l_department := department_tp.onerow (dept_in);
11   l_employees := employee_qp.ar_fk_emp_department (dept_in);
12
13   FOR l_index IN 1 .. l_employees.COUNT
14   LOOP
15     IF employee_rp.eligible_for_bonus (rec)
16     THEN
17       employee_cp.upd_onecol_pky
18         (colname_in    => 'salary'
19          , new_value_in => l_employees (l_index).salary
20            + bonus_in
21          , employee_id_in => l_employees (l_index).employee_id
22          , rows_out     => l_rows_updated
23        );
24     END IF;
25   END LOOP;
26
27   ... Дополнительная обработка имени и других элементов...
28 END;

```

В следующей таблице объясняются изменения, внесенные во второй версии.

| Строки | Описание |
|--------|---|
| 2–7 | Объявления, основанные на таблицах, уже не используют %TYPE и %ROWTYPE. Вместо них предоставляется «пакет типов» с объявлениями SUBTYPE, которые в свою очередь базируются на %TYPE и %ROWTYPE. При таком подходе коду приложения не нужно предоставлять прямой доступ к нижележащим таблицам (невозможный в инкапсулированной среде) |
| 10 | SELECT INTO заменяется вызовом функции, возвращающей «одну строку» информации для первичного ключа |
| 11 | Вызов функции, получающей все строки работников для внешнего ключа (идентификатор отдела). Функция использует BULK COLLECT и возвращает коллекцию записей, демонстрируя, как инкапсулированный код упрощает использование новых возможностей PL/SQL |
| 13–25 | Цикл FOR с курсором заменяется циклом FOR со счетчиком, перебирающим содержимое коллекции |
| 17–23 | Динамический SQL используется для обновления одного столбца с заданным первичным ключом |

В целом команды SQL были исключены из программы и заменены вызовами процедур и функций, предназначенных для многократного использования. В моем приложении такое решение оптимизирует SQL и способствует более эффективному написанию более надежного кода. Построение (или генерирование) таких пакетов ни в коей мере не является тривиальным делом, я и понимаю, что большинство читателей вряд ли захотят или смогут использовать «чистый» инкапсулированный подход. Однако многие преимущества инкапсуляции данных могут использоваться и без полной переработки стиля программирования. Как минимум я рекомендую:

- Скрыть все однострочные запросы за функциональным интерфейсом. Тем самым вы обеспечите гарантированную обработку ошибок и сможете выбрать оптимальную реализацию (например, неявные или явные курсоры).
- Определить, с какими таблицами разработчики чаще всего взаимодействуют напрямую, и построить для них прослойку программного кода.

- Создать пакетные программы для сложных транзакций. Если операция «добавление нового заказа» включает вставку двух строк, обновление шести строк и т. д., обязательно встройте эту логику в процедуру, которая скрывает все сложности. Не рассчитывайте на то, что разработчики сами во всем разберутся (и все правильно запрограммируют в разных местах!).

Исключение жесткого кодирования литералов

Практически в каждом приложении используются всевозможные «волшебные значения» — литералы, имеющие особый смысл для системы. Например, это могут быть коды типов или граничные значения для проверки данных. Конечно, пользователи скажут вам, что волшебные значения никогда не изменяются. «В моем балансе всегда будет *ровно* 25 позиций», — говорит один. «Родительская компания *всегда* будет называться ATLAS HQ», — клянется другой. Не верьте клятвам и никогда не фиксируйте их в своих программах. Возьмем следующие команды IF:

```
IF footing_difference BETWEEN 1 and 100
THEN
    adjust_line_item;
END IF;
```

```
IF cust_status = 'C'
THEN
    reopen_customer;
END IF;
```

Тот, кто пишет подобный код, сам напрашивается на неприятности. Ваша жизнь намного упростится, если вы создадите пакет с именованными константами:

```
PACKAGE config_pkg
IS
    closed_status      CONSTANT VARCHAR2(1) := 'C';
    open_status        CONSTANT VARCHAR2(1) := 'O';
    active_status      CONSTANT VARCHAR2(1) := 'A';
    inactive_status    CONSTANT VARCHAR2(1) := 'I';
    min_difference      CONSTANT PLS_INTEGER := 1;
    max_difference      CONSTANT PLS_INTEGER := 100;
    earliest_date       CONSTANT DATE := SYSDATE;
    latest_date        CONSTANT DATE := ADD_MONTHS (SYSDATE, 120);
END config_pkg;
```

С таким пакетом две предшествующие команды IF принимают следующий вид:

```
IF footing_difference
    BETWEEN config_pkg.min_difference and config_pkg.max_difference
THEN
    adjust_line_item;
END IF;

IF cust_status = config_pkg.closed_status
THEN
    reopen_customer;
END IF;
```

Если в будущем какое-либо из «волшебных значений» изменится, достаточно изменить соответствующую константу в конфигурационном пакете. Вносить изменения в остальных модулях не нужно. Практически в каждом приложении из тех, которые я рецензировал (и некоторые из тех, что я написал), ошибочно включались жестко закодированные «волшебные значения». В каждом случае разработчику приходилось вносить множественные изменения в программы — как в фазе разработки, так и в фазе сопровождения. Иногда это создает проблемы, иногда оборачивается сущим кошмаром;

я не могу выразить словами, насколько важно консолидировать все «волшебные значения» в одном или нескольких пакетах.

Другой пример такого пакета приведен в файле `utl_file_constants.pkg`. Этот пакет выбирает несколько иной путь: все значения скрываются в теле пакета. Спецификация пакета состоит только из функций, возвращающих значения. Если вдруг понадобится изменить какое-нибудь значение, перекомпилировать спецификацию пакета не придется — как и зависимые программы.

Наконец, если вам доведется выбирать литеральные значения, которые планируется скрыть за константами, старайтесь использовать невероятные значения, которые вряд ли будут использоваться как литералы. Предположим, процедура должна возвращать индикатор состояния: успех или неудача? Типичные значения таких флагов — 0 и 1, 5 и F и т. д. Тем не менее у таких значений есть один недостаток: они коротки и интуитивно понятны, поэтому у недисциплинированного программиста возникает соблазн «смухлеть» и напрямую использовать литералы в коде. Возьмем следующий пример:

```
PACKAGE do_stuff
IS
    c_success CONSTANT PLS_INTEGER := 0;
    c_failure CONSTANT PLS_INTEGER := 1;
    PROCEDURE big_stuff (stuff_key_in IN PLS_INTEGER, status_out OUT PLS_INTEGER);
END do_stuff;
```

Скорее всего, с таким определением вы столкнетесь с использованием `big_stuff` следующего вида:

```
do_stuff.big_stuff (l_stuff_key, l_status);
IF l_status = 0
THEN
    DBMS_OUTPUT.PUT_LINE ('Все нормально!');
END IF;
```

С другой стороны, если спецификация пакета выглядит так:

```
PACKAGE do_stuff
IS
    /* Произвольные литеральные значения! */
    c_success CONSTANT PLS_INTEGER := -90845367;
    c_failure CONSTANT PLS_INTEGER := 55338292;
    PROCEDURE big_stuff (stuff_key_in IN PLS_INTEGER, status_out OUT PLS_INTEGER);
END do_stuff;
```

вы *никогда* не увидите такой код:

```
do_stuff.big_stuff (l_stuff_key, l_status);
IF l_status = -90845367
THEN
    DBMS_OUTPUT.PUT_LINE ('Все нормально!');
END IF;
```

Он выглядит просто неприлично.

Устранение недостатков встроенных функций

Некоторые стандартные пакеты Oracle — такие, как `UTL_FILE` или `DBMS_OUTPUT`, — либо содержат неприятные ошибки, либо отражают неудачные решения, принятые проектировщиками. У всех нас есть «больные мозоли», и они не всегда связаны с тем, какие вспомогательные средства создает для нас Oracle. Как насчет «гениального» консультанта, который явился в ваш городок в прошлом году? Вы все еще пытаетесь разгрести код, оставшийся после него? Даже если заменить этот код невозможно, вы по крайней мере можете построить свой пакет на основе чужих пакетов, неудачно спроектированных структур данных и т. д. и по возможности исправить чужие ошибки.

Вместо того чтобы заполнять страницы книги примерами, я перечислю файлы нескольких пакетов, размещенных на сайте книги. Эти примеры демонстрируют использование пакетов, а также ряд других полезных возможностей. Я рекомендую просмотреть файлы *.pkg на сайте — вы найдете в них код, который может пригодиться в ваших приложениях. С чего стоит начать?

- filepath.pkg — добавляет поддержку путей в UTL_FILE, чтобы поиск заданного файла мог проводиться по нескольким каталогам.
- xfile.pkg и JFile.java (или sf_file.pks/pkb и sf_file.java) — расширяет область применения UTL_FILE. Пакет на базе класса Java способен выполнять многие задачи, не поддерживаемые UTL_FILE. Пакет xfile также предоставляет стопроцентную поддержку интерфейса UTL_FILE. Это означает, что вы можете провести глобальный поиск с заменой «UTL_FILE» на «xfile» — и все будет работать как прежде!
- sf_out.pks/pkb, bpl.sp, do.pkg — замена функциональности вывода DBMS_OUTPUT, которая поможет обойти недостатки проектирования (невозможность вывода BOOLEAN или — до выхода Oracle Database 10g — строк длиной более 255 байт).

Группировка логически связанных функций

Если ваша программа содержит десяток процедур и функций, связанных с конкретной функциональностью или аспектом вашего приложения, разместите их в пакете, чтобы упростить управление этим кодом (и его поиск). Это особенно важно при программировании бизнес-правил приложения, в реализации которых следует соблюдать некоторые важные правила:

- Не пытайтесь жестко кодировать эти правила (обычно в нескольких местах) в компонентах приложения.
- Не распределяйте реализацию правил по нескольким автономным программам, которыми будет трудно управлять.

Прежде чем браться за построение приложения, сконструируйте набор пакетов, инкапсулирующих все его правила. Иногда эти правила являются частью большего пакета — например, пакета инкапсуляции таблиц. В других случаях можно создать пакет, который не содержит ничего, кроме ключевых правил, как в следующем примере:

```
/* Файл в Сети: custrules.pkg */
PACKAGE customer_rules
IS
    FUNCTION min_balance RETURN PLS_INTEGER;

    FUNCTION eligible_for_discount
        (customer_in IN customer%ROWTYPE)
        RETURN BOOLEAN;

    FUNCTION eligible_for_discount
        (customer_id_in IN customer.customer_id%TYPE)
        RETURN BOOLEAN;

END customer_rules;
```

Функция eligible_for_discount скрыта в пакете, чтобы ею было удобнее управлять. Я также использую перегрузку, чтобы предоставить два разных интерфейса к формуле: первый получает первичный ключ и проверяет клиента по базе данных, а второй применяет свою логику к информации клиента, уже загруженной в запись %ROWTYPE. Почему я так поступил? Потому что если оператор уже запросил информацию клиента из базы данных, он может воспользоваться перегруженной версией для %ROWTYPE и избежать второго запроса.

Конечно, не вся «логически связанная функциональность» имеет отношение к бизнес-правилам. Допустим, мне нужно расширить возможности встроенных функций PL/SQL для работы со строками. Вместо того чтобы создавать 12 разных функций, я создаю пакет, размещаю все функции в этом пакете и сообщаю другим разработчикам, как обратиться к этой функциональности.

Кэширование статических данных сеанса для ускорения работы приложения

Используйте данные пакетов для улучшения времени отклика приложения посредством кэширования статических данных (без повторных запросов). Это можно сделать на нескольких уровнях. Для каждого варианта в следующем списке я привожу несколько полезных примеров кода, доступных на сайте книги.

- Кэширование одного значения — например, имени текущего пользователя (полученного функцией `USER`). Примеры: `thisuser.pkg` и `thisuser.tst`.
- Кэширование одной строки или информационного набора — например, параметров конфигурации заданного пользователя. Примеры: `init.pkg` и `init.tst`.
- Кэширование списка значений — например, содержимого статической таблицы подстановки. Примеры: `emplu.pkg` и `emplu.tst`.
- Использование файлов `.tst` для сравнения быстродействия с кэшированием и без него.

Кэширование на базе пакетов — всего лишь одна из разновидностей кэширования, доступных для разработчиков PL/SQL. За более подробным описанием всех возможностей кэширования обращайтесь к главе 21.



Если вы решите использовать кэширование уровня пакета, помните, что данные кэшируются по отдельности для каждого сеанса, обращающегося к пакету (в глобальной области PGA). Таким образом, если кэш занимает 2 Мбайт и в системе существует 1000 одновременно подключенных сеансов, вы расходуете 2 Гбайт памяти своей системы — кроме всей остальной памяти, расходуемой базой данных.

Пакеты и объектные типы

Пакеты представляют собой контейнеры, объединяющие данные и элементы кода. Объектные типы тоже представляют собой контейнеры, объединяющие данные и элементы кода. Нужны ли два типа контейнеров, не дублируют ли они функции друг друга? Не вытесняются ли пакеты объектными типами, особенно после того, как в Oracle появилась поддержка наследования? И если нет, то в каких случаях следует использовать объектные типы, а в каких — пакеты?

Пакеты и объектные типы действительно имеют много общего: и те и другие могут содержать одну или несколько программ и структур данных, а также спецификацию и тело. Однако между ними есть и принципиальные различия:

- Объектный тип — это *шаблон* данных, на основе которого можно создавать любое количество экземпляров типа (объектов). Каждый экземпляр имеет полный набор атрибутов (данных) и методов (процедур и функций). Экземпляры объектных типов могут храниться в базе данных. Пакет же является самостоятельной структурой, подобной статическому объектному типу, и создавать его экземпляры нельзя.
- Объектные типы поддерживают наследование. Это означает, что можно объявить объектный подтип, который *унаследует* все атрибуты и методы супертипа. У паке-

тов концепция иерархии и наследования не существует. Подробнее об объектной архитектуре рассказывается в главе 26.

- Пакеты позволяют создавать приватные, скрытые данные и программы. Объектные типы не поддерживают скрытых методов и атрибутов — все их компоненты объявляются как общие и являются общедоступными (хотя реализация методов скрывается в теле пакета).

Так когда же следует использовать объектные типы, а когда — пакеты? В настоящее время лишь немногие разработчики используют объектные типы и объектно-реляционную модель Oracle. Для большинства пакеты все еще остаются основными строительными блоками приложений PL/SQL.

Если вы планируете применять объектные типы, рекомендуем поместить значительную часть сложного кода в пакеты и вызывать его из методов объектных типов. Это позволит сохранить гибкость кода, присущую объектному подходу, и обеспечить возможность совместного использования кода разными элементами приложения.

19

Триггеры

Триггеры — это именованные программные блоки, выполняемые в ответ на происходящие в базе данных события. Они относятся к числу важнейших элементов профессионально спроектированных приложений Oracle и обычно используются для выполнения следующих действий:

- **Проверка вносимых в таблицы изменений.** Поскольку логика проверки данных непосредственно связана с конкретным объектом базы данных, триггеры гарантируют ее строгое выполнение и соблюдение.
- **Автоматизация сопровождения базы данных.** Начиная с Oracle8i можно было использовать триггеры, автоматически выполняемые при загрузке и выгрузке базы данных, для выполнения операций инициализации и очистки. Это значительно удобнее, чем создавать для этих операций внешние по отношению к базе данных сценарии.
- **Точная настройка ограничений на выполнение административных операций.** При помощи триггеров можно проверить, допускается ли выполнение определенной операции над конкретным объектом базы данных (например, удаление или модификация таблицы). Когда правила проверки реализованы в виде триггеров, обойти их очень трудно, если вообще возможно.

Существуют пять видов событий, с которыми можно связывать триггеры:

- **Команды DML (Data Manipulation Language).** Триггеры DML запускаются в ответ на вставку, обновление и удаление строки таблицы базы данных. Их можно использовать с целью проверки значений, устанавливаемых по умолчанию, выполнения аудита изменений и даже запрета определенных команд DML.
- **Команды DDL (Data Definition Language).** Триггеры DDL запускаются в ответ на выполнение команд DDL — например, при создании таблицы. С их помощью можно выполнять аудит и запрещать определенные операции.
- **События базы данных.** Триггеры событий базы данных используются при запуске и остановке базы данных, при подключении и отключении сервера, а также при возникновении ошибок Oracle. Начиная с Oracle8i они также позволяют получать информацию об операциях с базой данных.
- **Триггеры INSTEAD OF.** Замещающие триггеры (триггеры INSTEAD OF) являются альтернативой триггерам DML. Они запускаются непосредственно перед операциями вставки, обновления, удаления, и их код определяет, какие действия следует выполнить вместо соответствующей операции DML. Триггеры INSTEAD OF управляют операциями над представлениями, но не над таблицами. С их помощью можно

преобразовывать необновляемые представления в обновляемые, изменяя при необходимости их поведение.

- **Приостановленные команды.** В Oracle9i введена концепция приостановленных команд. Если в ходе выполнения команды возникла проблема доступности пространства (недостаточно табличного пространства или исчерпана квота), Oracle может перевести ее в режим приостановления до тех пор, пока проблема не будет решена. С данным событием можно связать триггер, который автоматически уведомляет пользователя о проблеме или даже самостоятельно устраняет ее.

Все эти типы триггеров рассматриваются в данной главе. Для каждого из них приводится описание синтаксиса, примеры и рекомендации по применению.

Если вам потребуется эмулировать триггеры для команд `SELECT` (запросов), изучите механизм FGA (Fine-Grained Auditing), описанный в главе 23, а также в книге *Oracle PL/SQL for DBAs* (издательство O'Reilly).

Триггеры уровня команд DML

Триггеры уровня команд DML (или просто *триггеры DML*) активизируются после вставки, обновления или удаления строк конкретной таблицы (рис. 19.1). Это самый распространенный тип триггеров, особенно часто применяемый разработчиками. Остальные триггеры используются преимущественно администраторами базы данных. В Oracle11i появилась возможность объединения нескольких триггеров DML в один *составной триггер*.

Прежде чем браться за написание триггера, необходимо ответить на следующие вопросы:

- Как триггер будет запускаться — по одному разу для каждой команды SQL или для каждой модифицируемой ею строки?
- Когда именно должен вызываться создаваемый триггер — до или после выполнения операции над строками?
- Для каких операций должен срабатывать триггер — вставки, обновления, удаления или их определенной комбинации?

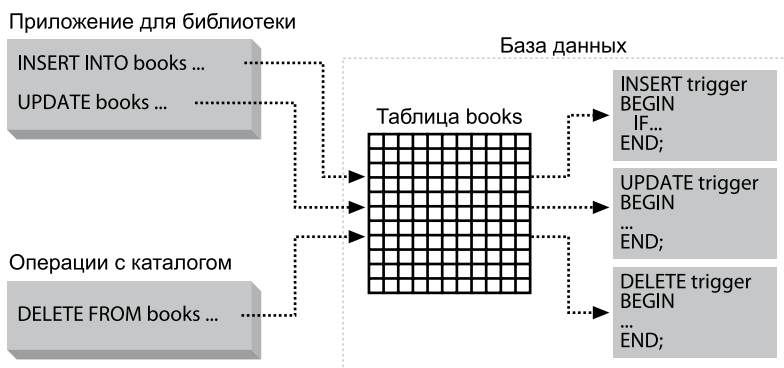


Рис. 19.1. Схема срабатывания триггеров DML

Основные концепции триггеров

Прежде чем переходить к синтаксису и примерам использования триггеров DML, следует познакомиться с их концепциями и терминологией.

- **Триггер BEFORE.** Вызывается до внесения каких-либо изменений (например, BEFORE INSERT).
- **Триггер AFTER.** Выполняется для отдельной команды SQL, которая может обрабатывать одну или более записей базы данных (например, AFTER UPDATE).
- **Триггер уровня команды.** Выполняется для команды SQL в целом (которая может обрабатывать одну или несколько строк базы данных).
- **Триггер уровня записи.** Выполняется для отдельной записи, обрабатываемой командой SQL. Если, предположим, таблица `books` содержит 1000 строк, то следующая команда UPDATE модифицирует все эти строки:

```
UPDATE books SET title = UPPER (title);
```

И если для таблицы определен триггер уровня записи, он будет выполнен 1000 раз.

- **Псевдозапись NEW.** Структура данных с именем NEW так же выглядит и обладает (почти) такими же свойствами, как запись PL/SQL. Эта псевдозапись доступна только внутри триггеров обновления и вставки; она содержит значения модифицированной записи после внесения изменений.
- **Псевдозапись OLD.** Структура данных с именем OLD так же выглядит и обладает (почти) такими же свойствами, как запись PL/SQL. Эта псевдозапись доступна только внутри триггеров обновления и вставки; она содержит значения модифицированной записи до внесения изменений.
- **Секция WHEN.** Часть триггера DML, определяющая условия выполнения кода триггера (и позволяющая избежать лишних операций).

Примеры сценариев с использованием триггеров DML

На сайте книги размещены примеры сценариев, демонстрирующих работу описанных в предыдущем разделе типов триггеров.

| Тип триггера | Файлы | Описание |
|---|--------------------------|---|
| Триггеры уровня команды и уровня записи | copy_tables.sql | Создает две идентичные таблицы: одну с данными, а другую пустую |
| | statement_vs_row.sql | Создает два простых триггера: один уровня команды, а другой уровня записи. После выполнения этих сценариев выполните следующую команду и просмотрите результаты (в режиме SERVEROUTPUT ON, чтобы они выводились на экран): INSERT INTO to_table SELECT * FROM from_table; |
| Триггеры BEFORE и AFTER | before_vs_after.sql | Создает триггеры BEFORE и AFTER. После выполнения сценария выполните следующую команду и просмотрите результаты: INSERT INTO to_table SELECT * FROM from_table; |
| Триггеры для разных DML-операций | one_trigger_per_type.sql | Создает триггеры AFTER_INSERT, UPDATE и DELETE для таблицы to_table. После выполнения сценария выполните следующие команды и просмотрите результаты: INSERT INTO to_table VALUES (1); UPDATE to_table SET col1 10; DELETE to_table; |

Триггеры в транзакциях

По умолчанию триггеры DML участвуют в транзакциях, из которых они запущены. Это означает, что:

- если триггер инициирует исключение, будет выполнен откат соответствующей части транзакции;
- если триггер сам выполнит команду DML (например, вставит запись в таблицу-журнал), она станет частью главной транзакции;
- в триггере DML нельзя выполнять команды COMMIT и ROLLBACK.



Если триггер DML определен как автономная транзакция (см. главу 14), то все команды DML, выполняемые внутри триггера, будут сохраняться или отменяться (командой COMMIT или ROLLBACK) независимо от основной транзакции.

В следующем разделе описан синтаксис объявления триггера DML и приведен пример, в котором использованы многие компоненты и параметры триггеров этого типа.

Создание триггера DML

Команда создания (или замены) триггера DML имеет следующий синтаксис:

```

1  CREATE [OR REPLACE] TRIGGER имя_триггера
2  {BEFORE | AFTER}
3  {INSERT | DELETE | UPDATE | UPDATE OF список_столбцов } ON имя_таблицы
4  [FOR EACH ROW]
5  [WHEN (...)]
6  [DECLARE ... ]
7  BEGIN
8  ...исполняемые команды...
9  [EXCEPTION ... ]
10 END [имя_триггера];

```

Описание всех перечисленных элементов приведено в таблице.

| Строки | Описание |
|--------|--|
| 1 | Создание триггера с заданным именем. Секция OR REPLACE не обязательна. Если триггер существует, а секция REPLACE отсутствует, попытка создания триггера приведет к ошибке ORA-4081. Вообще говоря, триггер и таблица могут иметь одинаковые имена (а также триггер и процедура), но мы рекомендуем использовать схемы выбора имен, предотвращающие подобные совпадения с неизбежной путаницей |
| 2 | Задание условий запуска триггера: до (BEFORE) или после (AFTER) выполнения команды либо обработки строки |
| 3 | Определение команды DML, с которой связывается триггер: INSERT, UPDATE или DELETE. Обратите внимание: триггер, связанный с командой UPDATE, может быть задан для всей строки или только для списка столбцов, разделенных запятыми. Столбцы можно объединять оператором OR и задавать в любом порядке. Кроме того, в строке 3 определяется таблица, с которой связывается данный триггер. Помните, что каждый триггер DML должен быть связан с одной таблицей |
| 4 | Если задана секция FOR EACH ROW, триггер будет запускаться для каждой обрабатываемой командой строки. Но если эта секция отсутствует, по умолчанию триггер будет запускаться только по одному разу для каждой команды (то есть будет создан триггер уровня команды) |
| 5 | Необязательная секция WHEN, позволяющая задать логику для предотвращения лишних выполнений триггера |
| 6 | Необязательный раздел объявлений для анонимного блока, составляющего код триггера. Если объявлять локальные переменные не требуется, это ключевое слово может отсутствовать. Никогда не объявляйте псевдозаписи NEW и OLD — они создаются автоматически |

| Строки | Описание |
|--------|--|
| 7, 8 | Исполняемый раздел триггера. Он является обязательным и должен содержать как минимум одну команду |
| 9 | Необязательный раздел исключений. В нем перехватываются и обрабатываются исключения, инициируемые только в исполняемом разделе |
| 10 | Обязательная команда END. Для наглядности в нее можно включить имя триггера |

Рассмотрим пару примеров триггеров DML.

- Первый триггер выполняет несколько проверок при добавлении или изменении строки в таблице сотрудников. В нем содержимое полей псевдозаписи **NEW** передается отдельным программам проверки:

```

TRIGGER validate_employee_changes
  AFTER INSERT OR UPDATE
  ON employees
  FOR EACH ROW
BEGIN
  check_date (:NEW.hire_date);
  check_email (:NEW.email);
END;
```

- Следующий триггер, запускаемый перед вставкой данных, проверяет изменения, производимые в таблице **ceo_compensation**. Для сохранения новой строки таблицы аудита вне главной транзакции в нем используется технология автономных транзакций:

```

TRIGGER bef_ins_ceo_comp
  BEFORE INSERT
  ON ceo_compensation
  FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO ceo_comp_history
    VALUES (:NEW.name,
            :OLD.compensation, :NEW.compensation,
            'AFTER INSERT', SYSDATE);
  COMMIT;
END;
```

Предложение WHEN

Предложение **WHEN** предназначено для уточнения условий, при которых должен выполняться код триггера. В приведенном далее примере с его помощью мы указываем, что основной код триггера должен быть реализован только при изменении значения столбца **salary**:

```

TRIGGER check_raise
  AFTER UPDATE OF salary
  ON employees
  FOR EACH ROW
WHEN ((OLD.salary != NEW.salary) OR
      (OLD.salary IS NULL AND NEW.salary IS NOT NULL) OR
      (OLD.salary IS NOT NULL AND NEW.salary IS NULL))
BEGIN
  ...
```

Иными словами, если при обновлении записи пользователь по какой-то причине оставит **salary** текущее значение, триггер активизируется, но его основной код выполняться не будет. Проверяя это условие в предложении **WHEN**, можно избежать затрат, связанных с запуском соответствующего кода PL/SQL.



В файле `genwhen.sp` на сайте книги представлена процедура для генерирования секции `WHEN`, которая проверяет, что новое значение действительно отличается от старого.

В большинстве случаев секция `WHEN` содержит ссылки на поля псевдозаписей `NEW` и `OLD`. В эту секцию разрешается также помещать вызовы встроенных функций, что и делается в следующем примере, где с помощью функции `SYSDATE` ограничивается время вставки новых записей:

```
TRIGGER valid_when_clause
BEFORE INSERT ON frame
FOR EACH ROW
WHEN ( TO_CHAR(SYSDATE, 'HH24') BETWEEN 9 AND 17 )
...
```

При использовании `WHEN` следует соблюдать ряд правил:

- Все логические выражения всегда должны заключаться в круглые скобки. Эти скобки не обязательны в команде `IF`, но необходимы в секции `WHEN` триггера.
- Перед идентификаторами `OLD` и `NEW` *не должно* стоять двоеточие (:). В секции `WHEN` следует использовать только встроенные функции.
- Пользовательские функции и функции, определенные во встроенных пакетах (таких, как `DBMS_UTILITY`), в нем вызывать нельзя. Чтобы вызвать такую функцию, переместите соответствующую логику в начало исполняемого раздела триггера.



Предложение `WHEN` может использоваться только в триггерах уровня записи. Поместив его в триггер уровня команды, вы получите сообщение об ошибке компиляции (ORA-04077).

Работа с псевдозаписями `NEW` и `OLD`

При запуске триггера уровня записи ядро `PL/SQL` создает и заполняет две структуры данных, имеющие много общего с записями. Речь идет о псевдозаписях `NEW` и `OLD` (префикс «псевдо» указывает на то, что они не обладают всеми свойствами записей `PL/SQL`). В псевдозаписи `OLD` хранятся исходные значения обрабатываемой триггером записи, а в псевдозаписи `NEW` — новые. Их структура идентична структуре записи, объявленной с атрибутом `%ROWTYPE` и создаваемой на основе таблицы, с которой связан триггер.

Несколько правил, которые следует принимать во внимание при работе с псевдозаписями `NEW` и `OLD`:

- Для триггеров, связанных с командой `INSERT`, структура `OLD` не содержит данных, поскольку старого набора значений у операции вставки *нет*.
- Для триггеров, связанных с командой `UPDATE`, заполняются обе структуры, `OLD` и `NEW`. Структура `OLD` содержит исходные значения записи до обновления, а `NEW` — значения, которые будут содержаться в строке после обновления.
- Для триггеров, связанных с командой `DELETE`, заполняется только структура `OLD`, а структура `NEW` остается пустой, поскольку запись удаляется.
- Псевдозаписи `NEW` и `OLD` также содержат столбец `ROWID`, который в обеих псевдозаписях всегда заполняется одинаковыми значениями.
- Значения полей записи `OLD` изменять нельзя; попытка приведет к ошибке `ORA-04085`. Значения полей структуры `NEW` модифицировать можно.

- Структуры NEW и OLD нельзя передавать в качестве параметров процедурам или функциям, вызываемым из триггера. Разрешается передавать лишь их отдельные поля. В сценарии `gentrigrec.sp` содержится программа, которая генерирует код копирования данных NEW и OLD в записи, передаваемые в параметрах.

- В ссылках на структуры NEW и OLD в анонимном блоке триггера перед соответствующими ключевыми словами необходимо ставить двоеточие:

```
IF :NEW.salary > 10000 THEN...
```

- Над структурами NEW и OLD нельзя выполнять операции уровня записи. Например, следующая команда вызовет ошибку компиляции триггера:

```
BEGIN :new := NULL; END;
```

С помощью секции `REFERENCING` в триггере можно менять имена псевдозаписей данных; это помогает писать самодокументированный код, ориентированный на конкретное приложение. Пример:

```
/* Файл в Сети: full_old_and_new.sql */
TRIGGER audit_update
AFTER UPDATE
ON frame
REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
FOR EACH ROW
BEGIN
    INSERT INTO frame_audit
        (bowler_id,
         game_id,
         old_score,
         new_score,
         change_date,
         operation)

        VALUES (:after_cheat.bowler_id,
                :after_cheat.game_id,
                :prior_to_cheat.score,
                :after_cheat.score,
                SYSDATE,
                'UPDATE');
END;
```

Запустите файл сценария `full_old_and_new.sql` и проанализируйте поведение псевдозаписей NEW и OLD.

Идентификация команды DML в триггере

Oracle предоставляет набор функций (также называемых *операционными директивами*) для идентификации команды DML, вызвавшей запуск триггера:

- `INSERTING` — возвращает TRUE, если триггер запущен в ответ на вставку записи в таблицу, с которой он связан, и FALSE в противном случае.
- `UPDATING` — возвращает TRUE, если триггер запущен в ответ на обновление записи в таблице, с которой он связан, и FALSE в противном случае.
- `DELETING` — возвращает TRUE, если триггер запущен в ответ на удаление записи из таблицы, с которой он связан, и FALSE в противном случае.

Пользуясь этими директивами, можно создать один триггер, который объединяет действия для нескольких операций. Пример:

```
/* Файл в Сети: one_trigger_does_it_all.sql */
TRIGGER three_for_the_price_of_one
BEFORE DELETE OR INSERT OR UPDATE ON account_transaction
FOR EACH ROW
```

```

BEGIN
  -- Сохранение информации о пользователе, вставившем новую строку
  IF INSERTING
  THEN
    :NEW.created_by := USER;
    :NEW.created_date := SYSDATE;

    -- Сохранение информации об удалении с помощью специальной программы
  ELSIF DELETING
  THEN
    audit_deletion(USER,SYSDATE);

    -- Сохранение информации о пользователе, который последним обновлял строку
  ELSIF UPDATING
  THEN
    :NEW.UPDATED_BY := USER;
    :NEW.UPDATED_DATE := SYSDATE;
  END IF;
END;
```

Функция `UPDATING` имеет перегруженную версию, которой в качестве аргумента передается имя конкретного столбца. Перегрузка функций является удобным способом изоляции операций обновления отдельных столбцов.

```

/* Файл в Сети: overloaded_update.sql */
TRIGGER validate_update
BEFORE UPDATE ON account_transaction
FOR EACH ROW
BEGIN
  IF UPDATING ('ACCOUNT_NO')
  THEN
    errpkg.raise('Account number cannot be updated');
  END IF;
END;
```

В спецификации имени столбца игнорируется регистр символов. Имя столбца до запуска триггера не анализируется, и если в таблице, связанной с триггером, заданного столбца не оказывается, функция просто возвращает `FALSE`.



Операционные директивы можно вызывать из любого кода PL/SQL, а не только из триггеров. Однако значение `TRUE` они возвращают лишь при использовании в триггерах DML или вызываемых из них программах.

Пример триггера DML

Одной из идеальных областей для применения триггеров является аудит изменений. Допустим, Памела, хозяйка боулинга, стала получать жалобы на посетителей, которые жульничают со своими результатами. Памела недавно написала приложение для ведения счета в игре и теперь хочет усовершенствовать его, чтобы выявить нечестных игроков.

В приложении Памелы центральное место занимает таблица `frame`, в которой записывается результат конкретного фрейма конкретной партии конкретного игрока:

```

/* Файл в Сети: bowlerama_tables.sql */
TABLE frame
(bowler_id    NUMBER,
 game_id      NUMBER,
 frame_number NUMBER,
 strike       VARCHAR2(1) DEFAULT 'N',
 spare        VARCHAR2(1) DEFAULT 'N',
 score        NUMBER,
```

```
CONSTRAINT frame_pk
PRIMARY KEY (bowler_id, game_id, frame_number))
```

Памела дополняет таблицу `frame` версией, в которой сохраняются все значения «до» и «после», чтобы она могла сравнить их и выявить несоответствия:

```
TABLE frame_audit
(bowler_id    NUMBER,
 game_id      NUMBER,
 frame_number NUMBER,
 old_strike   VARCHAR2(1),
 new_strike   VARCHAR2(1),
 old_spare    VARCHAR2(1),
 new_spare    VARCHAR2(1),
 old_score    NUMBER,
 new_score    NUMBER,
 change_date  DATE,
 operation    VARCHAR2(6))
```

Для каждого изменения в таблице `frame` Памела хочет отслеживать состояние строки до и после изменения. Она создает простой триггер:

```
/* Файл в Сети: bowlerama_full_audit.sql */
1  TRIGGER audit_frames
2  AFTER INSERT OR UPDATE OR DELETE ON frame
3  FOR EACH ROW
4  BEGIN
5      IF INSERTING THEN
6          INSERT INTO frame_audit(bowler_id,game_id,frame_number,
7                                  new_strike,new_spare,new_score,
8                                  change_date,operation)
9          VALUES(:NEW.bowler_id,:NEW.game_id,:NEW.frame_number,
10                 :NEW.strike,:NEW.spare,:NEW.score,
11                 SYSDATE,'INSERT');
12
13     ELSIF UPDATING THEN
14         INSERT INTO frame_audit(bowler_id,game_id,frame_number,
15                                 old_strike,new_strike,
16                                 old_spare,new_spare,
17                                 old_score,new_score,
18                                 change_date,operation)
19         VALUES(:NEW.bowler_id,:NEW.game_id,:NEW.frame_number,
20                 :OLD.strike,:NEW.strike,
21                 :OLD.spare,:NEW.spare,
22                 :OLD.score,:NEW.score,
23                 SYSDATE,'UPDATE');
24
25     ELSIF DELETING THEN
26         INSERT INTO frame_audit(bowler_id,game_id,frame_number,
27                                 old_strike,old_spare,old_score,
28                                 change_date,operation)
29         VALUES(:OLD.bowler_id,:OLD.game_id,:OLD.frame_number,
30                 :OLD.strike,:OLD.spare,:OLD.score,
31                 SYSDATE,'DELETE');
32     END IF;
33 END audit_frames;
```

В секции `INSERTING` (строки 6–11) для заполнения строки аудита используется псевдозапись `NEW`. Для `UPDATING` (строки 14–23) используется сочетание информации `NEW` и `OLD`. Для `DELETING` (строки 26–31) доступна только информация `OLD`. Памела создает триггер и ждет результатов.

Конечно, она не распространяется о своей новой системе. Салли — амбициозный, но не очень искусный игрок — понятия не имеет, что ее действия могут отслеживаться. Салли решает, что в этом году она должна стать чемпионом, и она не остановится ни перед

чем. У нее есть доступ к SQL*Plus, и она знает, что ее идентификатор игрока равен 1. Салли располагает достаточной информацией, чтобы полностью обойти графический интерфейс, подключиться к SQL*Plus и пустить в ход свое беспринципное «волшебство».

Салли сходу выписывает себе страйк в первом фрейме:

```
SQL> INSERT INTO frame
  2  (BOWLER_ID,GAME_ID,FRAME_NUMBER,STRIKE)
  3  VALUES(1,1,1,'Y');
1 row created.
```

Но затем она решает умерить аппетит и понижает результат первого фрейма, чтобы вызвать меньше подозрений:

```
SQL> UPDATE frame
  2  SET strike = 'N',
  3      spare = 'Y'
  4  WHERE bowler_id = 1
  5      AND game_id = 1
  6      AND frame_number = 1;
1 row updated.
```

Но что это? Салли слышит шум в коридоре. Она теряет самообладание и пытается замести следы:

```
SQL> DELETE frame
  2  WHERE bowler_id = 1
  3      AND game_id = 1
  4      AND frame_number = 1;
1 row deleted.
```

```
SQL> COMMIT;
Commit complete.
```

Она даже убеждается в том, что ее исправления были удалены:

```
SQL> SELECT * FROM frame;
no rows selected
```

Вытирая пот со лба, Салли завершает сеанс, но рассчитывает вернуться и реализовать свои планы.

Бдительная Памела подключается к базе данных и моментально обнаруживает, что пыталась сделать Салли. Для этого она выдает запрос к таблице аудита (кстати, Памела также может настроить ежечасный запуск задания через DBMS_JOB для автоматизации этой части аудита):

```
SELECT bowler_id, game_id, frame_number
       , old_strike, new_strike
       , old_spare, new_spare
       , change_date, operation
FROM frame_audit
```

Результат:

| BOWLER_ID | GAME_ID | FRAME_NUMBER | O | N | O | N | CHANGE_DA | OPERAT |
|-----------|---------|--------------|---|---|---|---|-----------|--------|
| 1 | 1 | 1 | Y | N | | | 12-SEP-00 | INSERT |
| 1 | 1 | 1 | Y | N | N | Y | 12-SEP-00 | UPDATE |
| 1 | 1 | 1 | N | | N | | 12-SEP-00 | DELETE |

Салли поймана с поличным! Из записей аудита прекрасно видно, что она пыталась сделать, хотя в таблице frame никаких следов не осталось. Все три команды — исходная вставка записи, понижение результата и последующее удаление записи — были перехвачены триггером DML.

Применение секции WHEN

После того как система аудита успешно проработала несколько месяцев, Памела принимает меры для дальнейшего устранения потенциальных проблем. Она просматривает интерфейсную часть своего приложения и обнаруживает, что изменяться могут только поля `strike`, `spare` и `score`. Следовательно, триггер может быть и более конкретным:

```
TRIGGER audit_update
  AFTER UPDATE OF strike, spare, score
  ON frame
  REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
  FOR EACH ROW
BEGIN
  INSERT INTO frame_audit (...)
    VALUES (...);
END;
```

Проходит несколько недель. Памеле не нравится ситуация, потому что новые записи создаются даже тогда, когда значения задаются равными сами себе. Обновления вроде следующего создают бесполезные записи аудита, которые показывают лишь отсутствие изменений:

```
SQL> UPDATE FRAME
      2 SET strike = strike;
      1 row updated.
SQL> SELECT old_strike,
      2         new_strike,
      3         old_spare,
      4         new_spare,
      5         old_score,
      6         new_score
      7 FROM frame_audit;

O N O N  OLD_SCORE  NEW_SCORE
- - - - -
Y Y N N
```

Триггер нужно дополнительно уточнить, чтобы он срабатывал только при фактическом изменении значений. Для этого используется секция `WHEN`:

```
/* Файл в Сети: final_audit.sql */
TRIGGER audit_update
  AFTER UPDATE OF STRIKE, SPARE, SCORE ON FRAME
  REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
  FOR EACH ROW
  WHEN ( prior_to_cheat.strike != after_cheat.strike OR
        prior_to_cheat.spare != after_cheat.spare OR
        prior_to_cheat.score != after_cheat.score )
BEGIN
  INSERT INTO FRAME_AUDIT ( ... )
    VALUES ( ... );
END;
```

Теперь данные будут появляться в таблице аудита только в том случае, если данные действительно изменились, а Памеле будет проще выявить возможных мошенников. Небольшая завершающая проверка триггера:

```
SQL> UPDATE frame
      2 SET strike = strike;
      1 row updated.

SQL> SELECT old_strike,
      2         new_strike,
```

продолжение ➤

```

3      old_spare,
4      new_spare,
5      old_score,
6      new_score
7  FROM frame_audit;
no rows selected

```

Использование псевдозаписей для уточнения триггеров

Памела реализовала в системе приемлемый уровень аудита; теперь ей хотелось бы сделать систему более удобной для пользователя. Самая очевидная идея — сделать так, чтобы система сама увеличивала счет во фреймах, заканчивающихся страйком или спэром, на 10. Это позволяет счетчику отслеживать счет только за последующие броски, а счет за страйк будет начисляться автоматически:

```

/* Файл в Сети: set_score.sql */
TRIGGER set_score
BEFORE INSERT ON frame
FOR EACH ROW
WHEN ( NEW.score IS NOT NULL )
BEGIN
    IF :NEW.strike = 'Y' OR :NEW.spare = 'Y'
    THEN
        :NEW.score := :NEW.score + 10;
    END IF;
END;

```



Помните, что значения полей в записях NEW могут изменяться только в BEFORE-триггерах строк.

Будучи человеком пунктуальным, Памела решает добавить проверку счета в ее набор триггеров:

```

/* File on Сети: validate_score.sql */
TRIGGER validate_score
AFTER INSERT OR UPDATE
ON frame
FOR EACH ROW
BEGIN
    IF :NEW.strike = 'Y' AND :NEW.score < 10
    THEN
        RAISE_APPLICATION_ERROR (
            -20001,
            'ERROR: Score For Strike Must Be >= 10'
        );
    ELSIF :NEW.spare = 'Y' AND :NEW.score < 10
    THEN
        RAISE_APPLICATION_ERROR (
            -20001,
            'ERROR: Score For Spare Must Be >= 10'
        );
    ELSIF :NEW.strike = 'Y' AND :NEW.spare = 'Y'
    THEN
        RAISE_APPLICATION_ERROR (
            -20001,
            'ERROR: Cannot Enter Spare And Strike'
        );
    END IF;
END;

```

Теперь любая попытка ввести строку, нарушающую это условие, будет отклонена:

```
SQL> INSERT INTO frame VALUES(1,1,1,'Y',NULL,5);
  2 INSERT INTO frame
    *
ERROR at line 1:
ORA-20001: ERROR: Score For Strike Must >= 10
```

Однотипные триггеры

Oracle позволяет связать с таблицей базы данных несколько триггеров одного типа. Рассмотрим такую возможность на примере, связанном с игрой в гольф. Следующий триггер уровня строки вызывается при вставке в таблицу новой записи и добавляет в нее комментарий, текст которого определяется соотношением текущего счета с номинальным значением 72:

```
/* Файл в Сети: golf_commentary.sql */
TRIGGER golf_commentary
  BEFORE INSERT
  ON golf_scores
  FOR EACH ROW
DECLARE
  c_par_score    CONSTANT PLS_INTEGER := 72;
BEGIN
  :new.commentary :=
    CASE
      WHEN :new.score < c_par_score THEN 'Under'
      WHEN :new.score = c_par_score THEN NULL
      ELSE 'Over' || ' Par'
    END;
END;
```

Эти же действия можно выполнить и с помощью трех отдельных триггеров уровня строки типа BEFORE INSERT с взаимоисключающими условиями, задаваемыми в секциях WHEN:

```
TRIGGER golf_commentary_under_par
  BEFORE INSERT ON golf_scores
  FOR EACH ROW
  WHEN (NEW.score < 72)
  BEGIN
    :NEW.commentary := 'Under Par';
  END;
```

```
TRIGGER golf_commentary_par
  BEFORE INSERT ON golf_scores
  FOR EACH ROW
  WHEN (NEW.score = 72)
  BEGIN
    :NEW.commentary := 'Par';
  END;
```

```
TRIGGER golf_commentary_over_par
  BEFORE INSERT ON golf_scores
  FOR EACH ROW
  WHEN (NEW.score > 72)
  BEGIN
    :NEW.commentary := 'Over Par';
  END;
```

Обе реализации абсолютно допустимы, и каждая обладает своими достоинствами и недостатками. Решение с одним триггером проще в сопровождении, поскольку весь код сосредоточен в одном месте, а решение с несколькими триггерами сокращает время синтаксического анализа и выполнения при необходимости более сложной обработки.

Очередность вызова триггеров

До выхода Oracle11g порядок срабатывания нескольких триггеров DML был непредсказуемым. В рассмотренном примере он несущественен, но как показывает следующий пример, в других ситуациях могут возникнуть проблемы. Какой результат будет получен для последнего запроса?

```
/* Файл в Сети: multiple_trigger_seq.sql */
TABLE incremented_values
(value_inserted    NUMBER,
 value_incremented NUMBER);

TRIGGER increment_by_one
BEFORE INSERT ON incremented_values
FOR EACH ROW
BEGIN
    :NEW.value_incremented := :NEW.value_incremented + 1;
END;

TRIGGER increment_by_two
BEFORE INSERT ON incremented_values
FOR EACH ROW
BEGIN
    IF :NEW.value_incremented > 1 THEN
        :NEW.value_incremented := :NEW.value_incremented + 2;
    END IF;
END;

INSERT INTO incremented_values
VALUES(1,1);
SELECT *
FROM incremented_values;
SELECT *
FROM incremented_values;
```

Есть какие-нибудь предположения? Для моей базы данных результаты получились такими:

```
SQL> SELECT *
      2    FROM incremented_values;

VALUE_INSERTED VALUE_INCREMENTED
-----
1                2
```

Это означает, что первым сработал триггер `increment_by_two`, который не выполнил никаких действий, потому что значение столбца `value_incremented` не превышало 1; затем сработал триггер `increment_by_one`, увеличивший значение столбца `value_incremented` на 1. А вы тоже получите такой результат? Вовсе не обязательно. Будет ли этот результат всегда одним и тем же? Опять-таки, ничего нельзя гарантировать. До выхода Oracle11g в документации Oracle было явно указано, что порядок запуска одностипных триггеров, связанных с одной таблицей, не определен и произволен, поэтому задать его явно невозможно. На этот счет существуют разные теории, среди которых наиболее популярны две: триггеры запускаются в порядке, обратном порядку их создания или же в соответствии с идентификаторами их объектов, но полагаться на такие предположения не стоит.

Начиная с Oracle11g гарантированный порядок срабатывания триггеров может определяться при помощи условия `FOLLOWS`, как показано в следующем примере:

```
TRIGGER increment_by_two
BEFORE INSERT ON incremented_values
FOR EACH ROW
```



```

FOLLOWS increment_by_one
BEGIN
  IF :new.value_incremented > 1 THEN
    :new.value_incremented := :new.value_incremented + 2;
  END IF;
END;

```

Теперь этот триггер заведомо будет активизирован раньше триггера `increment_by_one`. Тем самым гарантируется и результат вставки:

```

SQL> INSERT INTO incremented_values
2  VALUES(1,1);
1 row created.
SQL> SELECT *
2  FROM incremented_values;
VALUE_INSERTED VALUE_INCREMENTED
-----
1 4

```

Триггер `increment_by_one` увеличил вставленное значение до 2, а триггер `increment_by_two` увеличил его до 4. Такое поведение гарантировано, потому что оно определяется на уровне самого триггера — нет необходимости полагаться на догадки и предположения. Связи последовательности триггеров можно просмотреть в представлении зависимостей словаря данных Oracle:

```

SQL> SELECT referenced_name,
2  referenced_type,
3  dependency_type
4  FROM user_dependencies
5  WHERE name = 'INCREMENT_BY_TWO'
6  AND referenced_type = 'TRIGGER';
REFERENCED_NAME REFERENCED_TYPE DEPE
-----
INCREMENT_BY_ONE TRIGGER REF

```

Несмотря на поведение, описанное выше для Oracle Database 11g, при попытке откомпилировать триггер, следующий за неопределенным триггером, выводится сообщение об ошибке:

```

Trigger "SCOTT"."BLIND_FOLLOWER" referenced in FOLLOWS or PRECEDES clause may not exist

```

Ошибки при изменении таблицы

Изменяющиеся объекты трудно анализировать и оценивать. Поэтому когда триггер уровня строки пытается прочитать или изменить данные в таблице, находящейся в состоянии изменения (с помощью команды `INSERT`, `UPDATE` или `DELETE`), происходит ошибка с кодом `ORA-4091`.

В частности, эта ошибка встречается тогда, когда триггер уровня строк пытается выполнить чтение или запись в таблицу, для которой сработал триггер. Предположим, для таблицы сотрудников требуется задать ограничение на значения в столбцах, которое заключается в том, что при повышении оклада сотрудника его новое значение не должно превышать следующее значение по отделу более чем на 20%.

Казалось бы, для проверки этого условия можно использовать следующий триггер:

```

TRIGGER brake_on_raises
BEFORE UPDATE OF salary ON employee
FOR EACH ROW
DECLARE
  l_curr_max NUMBER;

```

продолжение ➤

```

BEGIN
  SELECT MAX (salary) INTO l_curr_max
  FROM employee;
  IF l_curr_max * 1.20 < :NEW.salary
  THEN
    errpkg.RAISE (
      employee_rules.en_salary_increase_too_large,
      :NEW.employee_id,
      :NEW.salary
    );
  END IF;
END;

```

Однако при попытке удвоить, скажем, оклад программиста PL/SQL, Oracle выдаст сообщение об ошибке:

```
ORA-04091: table SCOTT.EMPLOYEE is mutating, trigger/function may not see it
```

Тем не менее некоторые приемы помогут предотвратить выдачу этого сообщения об ошибке:

- В общем случае триггер уровня строки не может считывать или записывать данные таблицы, с которой он связан. Но подобное ограничение относится только к триггерам уровня строки. Триггеры уровня команд могут и считывать, и записывать данные своей таблицы, что дает возможность произвести необходимые действия.
- Если триггер выполняется как автономная транзакция (директива `PRAGMA AUTONOMOUS TRANSACTION` и выполнение `COMMIT` в теле триггера), тогда в нем можно *запрашивать* содержимое таблицы. Однако модификация такой таблицы все равно будет запрещена.

С каждым выпуском Oracle проблема ошибок изменения таблицы становится все менее актуальной, поэтому мы не станем приводить полное описание. На сайте книги размещен демонстрационный сценарий `mutation_zone.sql`. Кроме того, в файле `mutating_template.sql` представлен пакет, который может послужить шаблоном для создания вашей собственной реализации перевода логики уровня записей на уровень команд.

Составные триггеры

По мере создания триггеров, содержащих все больший объем бизнес-логики, становится трудно следить за тем, какие триггеры связаны с теми или иными правилами и как триггеры взаимодействуют друг с другом. В предыдущем разделе было показано, как три типа команд DML (вставка, обновление, удаление) объединяются в одном триггере, но разве не удобно было бы разместить триггеры строк и команд вместе в одном объекте кода?

В Oracle Database 11g появилась возможность использования составных триггеров для решения этой задачи. Следующий простой пример демонстрирует этот синтаксис:

```

/* Файл в Сети: compound_trigger.sql */
1  TRIGGER compounder
2  FOR UPDATE OR INSERT OR DELETE ON incremented_values
3  COMPOUND TRIGGER
4
5    v_global_var NUMBER := 1;
6
7  BEFORE STATEMENT IS
8  BEGIN
9    DBMS_OUTPUT.PUT_LINE('Compound:BEFORE S:' || v_global_var);
10   v_global_var := v_global_var + 1;
11  END BEFORE STATEMENT;
12
13  BEFORE EACH ROW IS

```

```

14 BEGIN
15     DBMS_OUTPUT.PUT_LINE('Compound:BEFORE R:' || v_global_var);
16     v_global_var := v_global_var + 1;
17 END BEFORE EACH ROW;
18
19 AFTER EACH ROW IS
20 BEGIN
21     DBMS_OUTPUT.PUT_LINE('Compound:AFTER R:' || v_global_var);
22     v_global_var := v_global_var + 1;
23 END AFTER EACH ROW;
24
25 AFTER STATEMENT IS
26 BEGIN
27     DBMS_OUTPUT.PUT_LINE('Compound:AFTER S:' || v_global_var);
28     v_global_var := v_global_var + 1;
29 END AFTER STATEMENT;
30
31 END;
```

Сходство с пакетами

Составные триггеры похожи на пакеты PL/SQL, не правда ли? Весь сопутствующий код и логика находятся в одном месте, что упрощает его отладку и изменение. Рассмотрим синтаксис более подробно.

Самое очевидное изменение — конструкция **COMPOUND TRIGGER**, сообщающая Oracle, что триггер содержит несколько триггеров, которые должны срабатывать вместе.

Следующее (и пожалуй, самое долгожданное) изменение встречается в строке 5: глобальная переменная! Наконец-то глобальные переменные могут определяться вместе с кодом, который с ними работает, — специальные пакеты для них больше не нужны:

```

PACKAGE BODY yet_another_global_package AS
    v_global_var NUMBER := 1;
    PROCEDURE reset_global_var IS
    ...
END;
```

В остальном синтаксис составных триггеров очень похож на синтаксис автономных триггеров, но не так гибок:

- **BEFORE STATEMENT** — код этого раздела выполняется до команды DML, как и в случае с автономным триггером **BEFORE**.
- **BEFORE EACH ROW** — код этого раздела выполняется перед обработкой каждой строки командой DML.
- **AFTER EACH ROW** — код этого раздела выполняется после обработки каждой строки командой DML.
- **AFTER STATEMENT** — код этого раздела выполняется после команды DML, как и в случае с автономным триггером **AFTER**.

Правила автономных триггеров также применимы и к составным триггерам — например, значения записей (**OLD** и **NEW**) не могут изменяться в триггерах уровня команд.

Различия с пакетами

Итак, составные триггеры похожи на пакеты, но означает ли это, что они так же работают? Нет — они работают лучше! Рассмотрим следующий пример:

```

SQL> BEGIN
2     insert into incremented_values values(1,1);
3     insert into incremented_values values(2,2);
4 END;
5 /
```

```

Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER R:3
Compound:AFTER S:4
Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER R:3
Compound:AFTER S:4

```

PL/SQL procedure successfully completed.

Обратите внимание: при выполнении второй команды для глобальной переменной снова выводится 1. Это связано с тем, что область действия составного триггера ограничивается командой DML, которая его инициирует. После завершения этой команды составной триггер и его значения, хранящиеся в памяти, перестают существовать. Это обстоятельство упрощает логику.

Дополнительным преимуществом ограниченной области действия является упрощенная обработка ошибок. Чтобы продемонстрировать это обстоятельство, я определяю в таблице первичный ключ для последующего нарушения:

```

SQL> ALTER TABLE incremented_values
2   add constraint a_pk
3   primary key ( value_inserted );

```

Теперь вставим одну запись:

```

SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER R:3
Compound:AFTER S:4
1 row created.

```

Пока без сюрпризов. Но следующая команда INSERT выдает ошибку из-за нарушения нового первичного ключа:

```

SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
insert into incremented_values values(1,1)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.A_PK) violated

```

Следующая команда INSERT также снова выдает ошибку первичного ключа. Но в этом как раз ничего примечательного нет — примечательно то, что глобальная переменная была снова инициализирована значением 1 без написания дополнительного кода. Команда DML завершилась, составной триггер вышел из области действия, и со следующей командой все начинается заново:

```

SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
insert into incremented_values values(1,1)
*
ERROR at line 1:
ORA-00001: unique constraint (DRH.A_PK) violated

```

Теперь мне не нужно включать дополнительную обработку ошибок или пакеты только для сброса значений при возникновении исключения.

FOLLOWS с составными триггерами

Составные триггеры также могут использоваться с синтаксисом FOLLOWS:

```
TRIGGER follows_compounder
BEFORE INSERT ON incremented_values
FOR EACH ROW
FOLLOWS compounder
BEGIN
    DBMS_OUTPUT.PUT_LINE('Following Trigger');
END;
```

Результат:

```
SQL> INSERT INTO incremented_values
2 values(8,8);
Compound:BEFORE S:1
Compound:BEFORE R:2
Following Trigger
Compound:AFTER R:3
Compound:AFTER S:4
1 row created.
```

Конкретные триггеры, находящиеся внутри составного триггера, не могут определяться как срабатывающие после каких-либо автономных или составных триггеров.



Если автономный триггер определяется как следующий за составным триггером, который не содержит триггер, срабатывающий по той же команде или строке, секция FOLLOWS просто игнорируется.

Триггеры уровня DDL

Oracle позволяет определять триггеры, срабатывающие при выполнении команды DDL — проще говоря, любых команд SQL, создающих или модифицирующих объекты базы данных (таблицы, индексы и т. д.). Несколько примеров команд DDL: CREATE TABLE, ALTER INDEX, DROP TRIGGER — каждая из них создает, изменяет или удаляет объект базы данных.

Синтаксис создания триггеров команд DDL почти не отличается от синтаксиса триггеров DML. Они различаются лишь перечнем иницилирующих событий и тем, что триггеры DDL не связываются с конкретными таблицами.



Весьма специфический триггер INSTEAD OF CREATE TABLE, описанный в конце раздела, позволяет манипулировать со стандартным поведением события CREATE TABLE. Не все аспекты синтаксиса и использования триггеров, описанные далее, применимы к этому типу триггеров.

Создание триггера DDL

Команда создания (или замены) триггера DDL имеет следующий синтаксис:

```
1 CREATE [OR REPLACE] TRIGGER имя_триггера
2 {BEFORE | AFTER } { событие_DDL } ON {DATABASE | SCHEMA}
3 [WHEN (...)]
4 DECLARE
5 Объявления переменных
6 BEGIN
7 ...код триггера...
8 END;
```

Элементы триггера описаны в следующей таблице.

| Строки | Описание |
|--------|--|
| 1 | Создание триггера с заданным именем. Если триггер существует, а секция REPLACE отсутствует, попытка создания триггера приведет к ошибке ORA-4081 |
| 2 | Строка определяет, должен ли триггер запускаться до или после наступления заданного события DDL, а также должен ли он запускаться для всех операций в базе данных или только в текущей схеме. Секция INSTEAD OF поддерживается только в Oracle9i Release 1 и последующих версиях |
| 3 | Необязательное условие WHEN, позволяющее задать логику для предотвращения ненужного выполнения триггера |
| 4–7 | Образец тела триггера |

Приведем пример простого триггера, оповещающего о создании любых объектов:

```
/* Файл в Сети: uninformed_town_crier.sql */
SQL> CREATE OR REPLACE TRIGGER town_crier
  2 AFTER CREATE ON SCHEMA
  3 BEGIN
  4   DBMS_OUTPUT.PUT_LINE('I believe you have created something!');
  5 END;
  6 /
Trigger created.

SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE a_table
  2 (col1 NUMBER);
Table created.

SQL> CREATE INDEX an_index ON a_table(col1);
Index created.

SQL> CREATE FUNCTION a_function RETURN BOOLEAN AS
  2 BEGIN
  3   RETURN(TRUE);
  4 END;
  5 /
Function created.

SQL> /* Очистка буфера DBMS_OUTPUT */
SQL> BEGIN NULL; END;
  2 /
I believe you have created something!
I believe you have created something!
I believe you have created something!

PL/SQL procedure successfully completed.
```



Текст, возвращаемый встроенным пакетом DBMS_OUTPUT, не будет выводиться триггером DDL до успешного выполнения блока PL/SQL, даже если этот блок не выполняет никаких действий.

Однако сообщая о создании объекта, этот триггер не уточняет, что же именно создается. Но на самом деле триггеры DDL могут предоставлять и более полную информацию, ведь триггер можно переписать и таким образом:

```
/* Файл в Сети: informed_town_crier.sql */
SQL> CREATE OR REPLACE TRIGGER town_crier
  2 AFTER CREATE ON SCHEMA
  3 BEGIN
```

```

4  -- Использование атрибутов события для получения более полной информации
5  DBMS_OUTPUT.PUT_LINE('I believe you have created a ' ||
6                        ORA_DICT_OBJ_TYPE || ' called ' ||
7                        ORA_DICT_OBJ_NAME);
8  END;
9  /

```

Trigger created.

```

SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE a_table
2  col1 NUMBER);
Table created.

```

```

SQL> CREATE INDEX an_index ON a_table(col1);
Index created.

```

```

SQL> CREATE FUNCTION a_function RETURN BOOLEAN AS
2  BEGIN
3      RETURN(TRUE);
4  END;
5  /
Function created.

```

```

SQL> /*-- Очистка буфера DBMS_OUTPUT */

```

```

SQL> BEGIN NULL; END;
2 /
I believe you have created a TABLE called A_TABLE
I believe you have created a INDEX called AN_INDEX
I believe you have created a FUNCTION called A_FUNCTION

```

PL/SQL procedure successfully completed.

В этих примерах представлены два важнейших аспекта триггеров DDL: события, с которыми они связываются, и атрибуты событий, которые в них доступны.

События триггеров

Список событий, которые можно связать с триггерами DDL, приведен в табл. 19.1. Любой триггер DDL может вызываться как до (BEFORE), так и после (AFTER) наступления указанного события.

Таблица 19.1. События DLL

| Событие DDL | Описание |
|-------------------------|---|
| ALTER | Создание объекта базы данных командой SQL ALTER |
| ANALYZE | Анализ состояния объекта базы данных командой SQL ANALYZE |
| ASSOCIATE STATISTICS | Связывание статистики с объектом базы данных |
| AUDIT | Включение аудита базы данных командой SQL AUDIT |
| COMMENT | Создание комментария для объекта базы данных |
| CREATE | Создание объекта базы данных командой SQL CREATE |
| DDL | Любое из перечисленных событий |
| DISASSOCIATE STATISTICS | Удаление статистики, связанной с объектом базы данных |
| DROP | Удаление объекта базы данных командой SQL DROP |
| GRANT | Назначение прав командой SQL GRANT |
| NOAUDIT | Отключение аудита базы данных командой SQL NOAUDIT |
| RENAME | Переименование объекта базы данных командой SQL RENAME |
| REVOKE | Отмена прав командой SQL REVOKE |
| TRUNCATE | Очистка таблицы командой SQL TRUNCATE |

Как и триггеры DML, триггеры DDL запускаются, когда в заданной базе данных или схеме происходят связанные с ними события. Количество типов триггеров, которые могут быть определены в базе данных или схеме, не ограничено.

Атрибутные функции

Oracle предоставляет набор функций (определенных в пакете `DBMS_STANDARD`) для получения информации о причине запуска триггера DDL и других связанных с ним параметрах (например, имя удаляемой таблицы). Перечень этих атрибутных функций приведен в табл. 19.2, а примеры их использования — в следующих разделах.

Таблица 19.2. События триггеров DDL и атрибутные функции

| Функция | Что возвращает |
|---|--|
| <code>ORA_CLIENT_IP_ADDRESS</code> | IP-адрес клиента |
| <code>ORA_DATABASE_NAME</code> | Имя базы данных |
| <code>ORA_DES_ENCRYPTED_PASSWORD</code> | Пароль текущего пользователя, зашифрованный с использованием алгоритма DES |
| <code>ORA_DICT_OBJ_NAME</code> | Имя объекта базы данных, связанного с командой DDL, которая вызвала запуск триггера |
| <code>ORA_DICT_OBJ_NAME_LIST</code> | Количество обработанных командой объектов. В параметре <code>NAME_LIST</code> возвращается полный список этих объектов в виде коллекции типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_DICT_OBJ_OWNER</code> | Имя владельца объекта базы данных, связанного с командой DDL, которая вызвала запуск триггера |
| <code>ORA_DICT_OBJ_OWNER_LIST</code> | Количество обработанных командой объектов. В параметре <code>NAME_LIST</code> возвращается полный список имен этих объектов в виде коллекции типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_DICT_OBJ_TYPE</code> | Тип объекта базы данных, связанного с командой DDL, вызвавшей запуск триггера (например, <code>TABLE</code> или <code>INDEX</code>) |
| <code>ORA GRANTEE</code> | Количество пользователей, получивших привилегии. В аргументе <code>USER_LIST</code> содержится полный список этих пользователей в виде коллекции типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_INSTANCE_NUM</code> | Номер экземпляра базы данных |
| <code>ORA_IS_ALTER_COLUMN</code> | <code>TRUE</code> , если изменяется столбец, заданный параметром <code>COLUMN_NAME</code> ; <code>FALSE</code> в противном случае |
| <code>ORA_IS_CREATING_NESTED_TABLE</code> | <code>TRUE</code> , если создается вложенная таблица; <code>FALSE</code> в противном случае |
| <code>ORA_IS_DROP_COLUMN</code> | <code>TRUE</code> , если удаляется столбец, заданный параметром <code>COLUMN_NAME</code> ; <code>FALSE</code> в противном случае |
| <code>ORA_LOGIN_USER</code> | Имя пользователя Oracle, для которого запущен триггер |
| <code>ORA_PARTITION_POS</code> | Позиция команды SQL для корректной вставки секции <code>PARTITION</code> |
| <code>ORA_PRIVILEGE_LIST</code> | Количество предоставленных или отмененных привилегий. В аргументе <code>PRIVILEGE_LIST</code> содержится полный список привилегий в виде коллекции типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_REVOKEE</code> | Количество пользователей, лишенных привилегий. В аргументе <code>USER_LIST</code> содержится полный список этих пользователей в виде коллекции типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_SQL_TXT</code> | Количество строк в команде SQL, которая вызвала запуск триггера. Аргумент <code>SQL_TXT</code> возвращает каждую строку команды в виде аргумента типа <code>DBMS_STANDARD.ORA_NAME_LIST_T</code> |
| <code>ORA_SYSEVENT</code> | Тип события, вызвавшего запуск триггера DDL (например, <code>CREATE</code> , <code>DROP</code> или <code>ALTER</code>) |
| <code>ORA_WITH_GRANT_OPTION</code> | <code>TRUE</code> , если привилегии предоставлены конструкцией <code>GRANT</code> ; <code>FALSE</code> в противном случае |

Об атрибутных функциях необходимо дополнительно сказать следующее:

- Тип данных `ORA_NAME_LIST_T` определен в пакете `DBMS_STANDARD` так:

```
TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
```

Иными словами, это вложенная таблица строк, каждая из которых может содержать до 64 символов.

- События триггеров DDL и атрибутные функции также определены в пакете `DBMS_STANDARD`. Для каждой из функций этого пакета Oracle создает независимую функцию, добавляя к ее имени префикс `ORA_`, для чего при создании базы данных выполняется сценарий `$ORACLE_HOME/rdbms/dbmstrig.sql`. В некоторых версиях Oracle этот сценарий содержит ошибки, из-за которых независимые функции не видны или не выполняются. Если вы сомневаетесь в правильности определения этих элементов, попросите администратора базы данных проверить сценарий и внести необходимые исправления.
- Представление словаря данных `USER_SOURCE` не обновляется до срабатывания любых триггеров DDL, `BEFORE` и `AFTER`. Иначе говоря, вы не сможете использовать эти функции для реализации системы контроля версий «до и после», построенной исключительно в границах базы данных и основанной на триггерах.

Применение событий и атрибутов

Возможности триггеров DDL лучше всего продемонстрировать на примерах. Для начала рассмотрим триггер, который блокирует создание любых объектов базы данных:

```
TRIGGER no_create
  AFTER CREATE ON SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (
    -20000,
    'ERROR : Objects cannot be created in the production database.'
  );
END;
```

После его создания в базе данных не удастся создать ни один объект:

```
SQL> CREATE TABLE demo (col1 NUMBER);
```

```
*
```

```
ERROR at line 1:
```

```
ORA-20000: Objects cannot be created in the production database.
```

Сообщение об ошибке получилось кратким и не слишком содержательным. Произошел сбой, но какой именно? Хорошо бы включить в сообщение дополнительную информацию, например указать тип объекта, который пытался создать пользователь:

```
/* Файл в Сети: no_create.sql */
```

```
TRIGGER no_create
  AFTER CREATE ON SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (-20000,
    'Cannot create the ' || ORA_DICT_OBJ_TYPE ||
    ' named ' || ORA_DICT_OBJ_NAME ||
    ' as requested by ' || ORA_DICT_OBJ_OWNER ||
    ' in production.');
```

```
END;
```

С таким триггером попытка создать таблицу в базе данных приведет к выводу более подробного сообщения об ошибке:

```
SQL> CREATE TABLE demo (col1 NUMBER);
```

```
*
```

```
ERROR at line 1:
```

```
ORA-20000: Cannot create the TABLE named DEMO as requested by SCOTT in production
```

Можно было бы даже реализовать эту логику в виде триггера BEFORE и воспользоваться событием ORA_SYSEVENT:

```
TRIGGER no_create
BEFORE DDL ON SCHEMA
BEGIN
    IF ORA_SYSEVENT = 'CREATE'
    THEN
        RAISE_APPLICATION_ERROR (-20000,
            'Cannot create the ' || ORA_DICT_OBJ_TYPE ||
            ' named ' || ORA_DICT_OBJ_NAME ||
            ' as requested by ' || ORA_DICT_OBJ_OWNER);
    ELSIF ORA_SYSEVENT = 'DROP'
    THEN
        -- Логика операций DROP
        ...
    END IF;
END;
```

Какой столбец был изменен?

Для получения информации о том, какой столбец был изменен командой ALTER TABLE, можно воспользоваться функцией ORA_IS_ALTER_COLUMN. Пример:

```
/* Файл в Сети: preserve_app_cols.sql */
TRIGGER preserve_app_cols
AFTER ALTER ON SCHEMA
DECLARE
    -- Курсор для получения информации о столбцах таблицы
    CURSOR curs_get_columns (cp_owner VARCHAR2, cp_table VARCHAR2)
    IS
        SELECT column_name
        FROM all_tab_columns
        WHERE owner = cp_owner AND table_name = cp_table;
BEGIN
    -- Если была изменена таблица ...
    IF ora_dict_obj_type = 'TABLE'
    THEN
        -- Для каждого столбца в таблице...
        FOR v_column_rec IN curs_get_columns (
            ora_dict_obj_owner,
            ora_dict_obj_name
        )
        LOOP
            -- Является ли текущий столбец измененным?
            IF ORA_IS_ALTER_COLUMN (v_column_rec.column_name)
            THEN
                -- Отклонить изменения в "критическом" столбце
                IF mycheck.is_application_column (
                    ora_dict_obj_owner,
                    ora_dict_obj_name,
                    v_column_rec.column_name
                )
                THEN
                    CENTRAL_ERROR_HANDLER (
                        'FAIL',
                        'Cannot alter core application attributes'
                    );
                END IF; -- критическая таблица/столбец
            END IF; -- текущий столбец был изменен
        END LOOP; -- для каждого столбца в таблице
    END IF; -- таблица была изменена
END;
```

Попытки изменения критических атрибутов приложения становятся невозможными. Помните, что эта логика не будет работать, если триггер срабатывает при добавлении новых столбцов. При срабатывании триггера DDL информация о столбцах еще не видна в словаре данных.

Попытки удаления конкретных столбцов можно проверять следующим образом:

```
IF ORA_IS_DROP_COLUMN ('COL2')
THEN
    что-то сделать
ELSE
    сделать что-то другое!
END IF;
```



Функции `ORA_IS_DROP_COLUMN` и `ORA_IS_ALTER_COLUMN` не обращают внимания на то, к какой таблице присоединен столбец; они работают исключительно по имени столбца.

Списки, возвращаемые атрибутными функциями

Некоторые атрибутные функции возвращают данные двух типов: список элементов и количество элементов. Например, функция `ORA_GRANTEE` возвращает список и количество пользователей, которым предоставлены определенные права, а функция `ORA_PRIVILEGE_LIST` — список и количество предоставленных прав. Обычно обе эти функции применяются в триггерах `AFTER GRANT`. Пример использования этих функций представлен в файле `privs.sql` на сайте книги. Фрагмент кода этого примера:

```
/* Файл в Сети: what_privs.sql */
TRIGGER what_privs
AFTER GRANT ON SCHEMA
DECLARE
    v_grant_type      VARCHAR2 (30);
    v_num_grantees    BINARY_INTEGER;
    v_grantee_list     ora_name_list_t;
    v_num_privs        BINARY_INTEGER;
    v_priv_list        ora_name_list_t;
BEGIN
    -- Получение информации о типе с последующей выборкой списков.
    v_grant_type := ORA_DICT_OBJ_TYPE;
    v_num_grantees := ORA_GRANTEE (v_grantee_list);
    v_num_privs := ORA_PRIVILEGE_LIST (v_priv_list);

    IF v_grant_type = 'ROLE PRIVILEGE'
    THEN
        DBMS_OUTPUT.put_line (
            'The following roles/privileges were granted');

        -- Вывод привилегии для каждого элемента списка.
        FOR counter IN 1 .. v_num_privs
        LOOP
            DBMS_OUTPUT.put_line ('Privilege ' || v_priv_list (counter));
        END LOOP;
```

Триггер прекрасно подходит для получения подробной информации о правах пользователей и объектах базы данных, указанных в командах `GRANT`. Эту информацию также можно сохранить в базе данных, добавив журнал с подробными сведениями об изменениях.

```
SQL> GRANT DBA TO book WITH ADMIN OPTION;
Grant succeeded.
```

продолжение ➤

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE('Flush buffer');
The following roles/privileges were granted
      Privilege UNLIMITED TABLESPACE
      Privilege DBA
      Grant Recipient BOOK
Flush buffer

SQL> GRANT SELECT ON x TO system WITH GRANT OPTION;
Grant succeeded.

SQL> EXEC DBMS_OUTPUT.PUT_LINE('Flush buffer');
The following object privileges were granted
      Privilege SELECT
      On X with grant option
      Grant Recipient SYSTEM
Flush buffer
```

Можно ли удалить неудаляемое?

Я показал, что триггеры DDL могут использоваться для предотвращения выполнения определенного типа операций DDL с конкретными объектами или типами объектов. А если я создам триггер, который предотвращает DDL-операции DROP, а затем попытаюсь удалить сам триггер? Не появится ли триггер, который по сути невозможно удалить? К счастью, в Oracle этот сценарий был предусмотрен:

```
SQL> CREATE OR REPLACE TRIGGER undroppable
2  BEFORE DROP ON SCHEMA
3  BEGIN
4    RAISE_APPLICATION_ERROR(-20000,'You cannot drop me! I am invincible!');
5  END;

SQL> DROP TABLE employee;
*
ERROR at line 1:
ORA-20000: You cannot drop me! I am invincible!

SQL> DROP TRIGGER undroppable;
Trigger dropped.
```

При работе с подключаемыми базами данных (Oracle Database 12c и выше) можно вставить ключевое слово **PLUGGABLE** перед **DATABASE** в определении триггера. **DATABASE** (без **PLUGGABLE**) определяет триггер на корневом уровне. В мультиарендной (multitenant) контейнерной базе данных (CDB) только пользователь, подключившийся к корневому уровню, может создать триггер для всей базы данных. **PLUGGABLE DATABASE** определяет триггер для подключаемой базы данных, к которой вы подключены. Триггер срабатывает каждый раз, когда любой пользователь заданной базы данных или PDB инициирует активизирующее событие.

Триггер INSTEAD OF CREATE

Oracle предоставляет триггер **INSTEAD OF CREATE** для автоматической группировки данных таблиц. Для этого триггер должен перехватить выполняемую команду SQL, вставить в нее условие группировки, а затем выполнить при помощи функции **ORA_SQL_TXT**. Следующий пример показывает, как это делается.

```
/* Файл в Сети: io_create.sql */
TRIGGER io_create
  INSTEAD OF CREATE ON DATABASE
WHEN (ORA_DICT_OBJ_TYPE = 'TABLE')
DECLARE
  v_sql  VARCHAR2 (32767); -- Генерируемый код sql
```

```

v_sql_t ora_name_list_t; -- таблица sql
BEGIN
  -- Получение выполняемой команды SQL
  FOR counter IN 1 .. ora_sql_txt (v_sql_t)
  LOOP
    v_sql := v_sql || v_sql_t (counter);
  END LOOP;
  -- Для определения условия PARTITION будет вызвана
  -- функция magic_partition.
  v_sql :=
    SUBSTR (v_sql, 1, ora_partition_pos)
    || magic_partition_function
    || SUBSTR (v_sql, ora_partition_pos + 1);

  /* Вставить перед именем таблицы имя пользователя.
  | Комбинации CRLF заменяются пробелами.
  | Операция требует наличия явной привилегии CREATE ANY TABLE,
  | если только вы не переключились на модель AUTHID CURRENT_USER.
  */
  v_sql :=
    REPLACE (UPPER (REPLACE (v_sql, CHR (10), ' '))
      , 'CREATE TABLE '
      , 'CREATE TABLE ' || ora_login_user || '.'
    );

  -- Выполнение сгенерированной команды SQL
  EXECUTE IMMEDIATE v_sql;
END;
```

Теперь группировка таблиц будет осуществляться автоматически в соответствии с логикой функции `my_partition`.

Oracle предоставляет несколько способов группировки (диапазоны, хеширование) и логических моделей группировки (например, по первичному ключу или по уникальному ключу). Вы должны решить, какие из них должны использоваться в вашей функции группировки.

Если не включить только что приведенную секцию `WHEN`, при попытке создания объектов, отличных от таблиц, происходит ошибка:

```

ORA-00604: error occurred at recursive SQL level 1
ORA-30511: invalid DDL operation in system triggers
```

Кроме того, при попытке создания триггера `INSTEAD OF` для любой другой операции DDL, кроме `CREATE`, компилятор выдает сообщение об ошибке (ORA-30513).



Триггеры `INSTEAD OF` для операций DML (вставка, обновление и удаление) рассматриваются далее в этой главе. Эти триггеры используют некоторые элементы синтаксиса триггера `INSTEAD OF CREATE` для таблиц, но этим сходство между ними ограничивается.

Триггеры событий базы данных

Триггеры событий базы данных запускаются при возникновении событий уровня базы данных. Далее перечислены восемь таких событий (два из них появились в Oracle Database 12c) с указанием условия запуска:

- `STARTUP` — при открытии базы данных;
- `SHUTDOWN` — при нормальном закрытии базы данных;
- `SERVERERROR` — при возникновении ошибки Oracle;

- LOGON — при запуске сеанса Oracle;
- LOGOFF — при нормальном завершении сеанса Oracle;
- DB_ROLE_CHANGE — при назначении резервной базы данных первичной, или наоборот.
- AFTER_CLONE (Oracle Database 12c) — может указываться только в сочетании с PLUGGABLE DATABASE. После копирования (клонирования) подключаемой базы данных (PDB, Pluggable DataBase) база данных иницирует триггер в новой PDB и удаляет его. Если при выполнении триггера происходит ошибка, то операция копирования завершается неудачей.
- BEFORE UNPLUG (Oracle Database 12c) — может указываться только в сочетании с PLUGGABLE DATABASE. Перед отключением PDB база данных активизирует триггер в новой PDB и удаляет его. Если при выполнении триггера происходит ошибка, то операция отключения завершается неудачей.

Эти триггеры предоставляют полезные возможности для автоматизации процесса администрирования базы данных и точного управления ее функционированием.

Создание триггера события базы данных

Синтаксис создания этих триггеров очень похож на синтаксис триггеров DDL:

```

1 CREATE [OR REPLACE] TRIGGER имя_триггера
2 {BEFORE | AFTER} {событие} ON {DATABASE | SCHEMA}
3 DECLARE
4 Объявления переменных
5 BEGIN
6 ...Код...
7 END;
```

Не все события поддерживают оба атрибута, BEFORE и AFTER. Некоторые комбинации просто не имеют смысла:

- BEFORE STARTUP. Даже если такие триггеры можно было бы создавать, когда они должны запускаться? При попытке создания триггеров такого типа выводится сообщение об ошибке (ORA-30500).
- AFTER SHUTDOWN. Опять же, когда должны запускаться такие триггеры? При попытке создания таких триггеров выводится сообщение об ошибке (ORA-30501).
- BEFORE LOGON. Эти триггеры уж слишком предусмотрительны: «Кажется, кто-то собирается подключиться — давайте сделаем что-нибудь!» Но Oracle подходит к делу более реалистично и выводит сообщение об ошибке (ORA-30508).
- AFTER LOGOFF. «Пожалуйста, вернитесь! Не отключайтесь!» При попытке создания таких триггеров выводится сообщение об ошибке (ORA-30509).
- BEFORE SERVERERROR. Мечта каждого программиста! Только представьте:

```

CREATE OR REPLACE TRIGGER BEFORE_SERVERERROR
BEFORE SERVERERROR ON DATABASE
BEGIN
    diagnose_impending_error;          -- обнаруживаем будущую ошибку
    fix_error_condition;               -- исправляем ее причину
    continue_as_if_nothing_happened;  -- продолжаем работу
END;
```

К сожалению, так не бывает. Мечты разбиваются сообщением об ошибке (ORA-30500).

Триггер STARTUP

Триггеры типа STARTUP запускаются при загрузке базы данных. Это прекрасный момент для выполнения подготовительных работ — например, для фиксации объектов в общем пуле, чтобы они не удалялись в связи с продолжительным бездействием.



Чтобы создать триггер события STARTUP, пользователь должен обладать привилегией ADMINISTER DATABASE TRIGGER.

Пример создания триггера STARTUP:

```
CREATE OR REPLACE TRIGGER startup_pinner
AFTER STARTUP ON DATABASE
BEGIN
    pin_plsql_packages;
    pin_application_packages;
END;
```

Триггеры SHUTDOWN

Триггеры BEFORE SHUTDOWN запускаются перед закрытием базы данных. Они отлично подходят для сбора статистики о системе. Приведем пример создания триггера события SHUTDOWN:

```
CREATE OR REPLACE TRIGGER before_shutdown
BEFORE SHUTDOWN ON DATABASE
BEGIN
    gather_system_stats;
END;
```



Триггеры SHUTDOWN запускаются только при остановке базы данных в режиме NORMAL или IMMEDIATE. Они не выполняются при завершении работы в режиме ABORT и в результате сбоев.

Триггер LOGON

Триггеры AFTER LOGON запускаются в начале сеанса Oracle. Их удобно использовать для формирования контекста сеанса и выполнения других задач предварительной настройки. Пример триггера события LOGON:

```
TRIGGER after_logon
AFTER LOGON ON SCHEMA
DECLARE
    v_sql VARCHAR2(100) := 'ALTER SESSION ENABLE RESUMABLE ' ||
                          'TIMEOUT 10 NAME ' || '''' ||
                          'OLAP Session' || '''';
BEGIN
    EXECUTE IMMEDIATE v_sql;
    DBMS_SESSION.SET_CONTEXT('OLAP Namespace',
                             'Customer ID',
                             load_user_customer_id);
END;
```

Триггеры LOGOFF

Триггеры BEFORE LOGOFF запускаются при нормальном отсоединении пользователя от базы данных; в них удобно собирать статистику уровня сеанса. Пример создания триггера для события LOGOFF:

```
TRIGGER before_logoff
BEFORE LOGOFF ON DATABASE
BEGIN
    gather_session_stats;
END;
```

Триггеры SERVERERROR

Триггеры AFTER SERVERERROR запускаются в тот момент, когда Oracle генерирует какую-либо ошибку. Исключения составляют следующие ошибки:

- ORA-00600 — внутренняя ошибка Oracle;
- ORA-01034 — нет доступа к Oracle;
- ORA-01403 — данные не найдены;
- ORA-01422 — при выборке возвращается больше строк, чем указано в запросе;
- ORA-01423 — при попытке выборки дополнительных строк произошла ошибка;
- ORA-04030 — нехватка памяти.

Кроме того, триггеры AFTER SERVERERROR не запускаются, когда исключение инициируется *внутри* триггера (для предотвращения бесконечной рекурсии).

Триггеры AFTER SERVERERROR не предоставляют средств для исправления ошибки — они предназначены только для сохранения информации о ней. На этой базе можно построить достаточно мощные механизмы регистрации ошибок.

Oracle также предоставляет встроенные функции (также определенные в пакете DBMS_STANDARD) для получения информации о стеке ошибок, формируемом при возникновении исключения:

- ORA_SERVER_ERROR — возвращает номер ошибки Oracle в заданной позиции стека. Если в этой позиции ошибки нет, возвращает 0.
- ORA_IS_SERVERERROR — возвращает TRUE, если в стеке ошибок текущего исключения имеется ошибка с заданным номером.
- ORA_SERVER_ERROR_DEPTH — возвращает количество ошибок в стеке.
- ORA_SERVER_ERROR_MSG — возвращает полный текст сообщения об ошибке для заданной позиции. Если в этой позиции ошибки нет, возвращает NULL.
- ORA_SERVER_ERROR_NUM_PARAMS — возвращает количество параметров, связанных с сообщением об ошибке в заданной позиции. Если в этой позиции ошибки нет, возвращает 0.
- ORA_SERVER_ERROR_PARAM — возвращает значение параметра в заданной позиции. Если параметр не найден, возвращает NULL.

Примеры триггеров SERVERERROR

Рассмотрим несколько примеров использования функций SERVERERROR. Начнем с очень простого триггера, который просто сообщает о возникновении ошибки:

```
TRIGGER error_echo
AFTER SERVERERROR
ON SCHEMA
BEGIN
    DBMS_OUTPUT.PUT_LINE ('You experienced an error');
END;
```

При возникновении ошибки Oracle (и при условии, что параметр SERVEROUTPUT имеет значение ON) на экран будет выведено такое сообщение:

```
SQL> SET SERVEROUTPUT ON
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
You experienced an error
BEGIN DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A')); END;
```

*

```
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
ORA-06512: at line 1
```


Обратите внимание: сообщение об ошибке Oracle выводится после сообщения триггера. Благодаря этому информацию об ошибке можно получить и сохранить еще до того, как Oracle предпримет какие-либо действия.



Триггеры SERVERERROR автоматически изолируются в собственных автономных транзакциях (см. главу 14). В частности, это означает, что можно записать информацию об ошибке в таблицу-журнал и сохранить ее командой COMMIT без изменения состояния транзакции сеанса, в которой произошла ошибка.

Триггер `error_logger` гарантирует, что информация обо всех ошибках (кроме перечисленных ранее) будет автоматически записана в журнал. При этом не имеет значения, кем или чем была вызвана ошибка — пользователем, самими приложением или какой-то программой.

```
/* Файл в Сети: error_log.sql */
TRIGGER error_logger
AFTER SERVERERROR
ON SCHEMA
DECLARE

    v_errnum    NUMBER;           -- Номер ошибки Oracle
    v_now       DATE := SYSDATE;  -- текущее время

BEGIN

    -- Для каждой ошибки в стеке ...
    FOR e_counter IN 1..ORA_SERVER_ERROR_DEPTH LOOP

        -- Записать информацию об ошибке в журнал; фиксация транзакции
        -- не требуется, поскольку данная транзакция автономна.
        INSERT INTO error_log(error_id,
                               username,
                               error_number,
                               sequence,
                               timestamp)
        VALUES(error_seq.nextval,
               USER,
               ORA_SERVER_ERROR(e_counter),
               e_counter,
               v_now);

    END LOOP;  -- для каждой ошибки в стеке

END;
```

Помните, что все новые строки таблицы `error_log` будут зафиксированы к моменту выполнения оператора `END`, поскольку триггер выполняется в автономной транзакции. Пример использования этого триггера:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error

SQL> SELECT * FROM error_log;
```

| USERNAME | ERROR_NUMBER | SEQUENCE | TIMESTAMP |
|----------|--------------|----------|-----------|
| BOOK | 6502 | 1 | 04-JAN-02 |
| BOOK | 6512 | 2 | 04-JAN-02 |

Почему в таблице хранятся две ошибки, когда была инициирована всего одна ошибка? Дело в том, что стек ошибок, генерируемый базой данных, содержит ошибки ORA-06502 и ORA-06512, которые регистрируются в порядке их возникновения.

Если вы хотите быстро определить, присутствует ли в стеке определенная ошибка, без ручного разбора его содержимого, используйте вспомогательную функцию `ORA_IS_SERVERERROR`. Эта функция чрезвычайно полезна для контроля за конкретными ошибками, которые могут потребовать дополнительной обработки — например, пользовательских исключений. Код может выглядеть примерно так:

```
-- Специальная обработка пользовательских ошибок.
-- Ошибки с 20000 по 20010 инициируются вызовами
-- RAISE_APPLICATION_ERROR
```

```
FOR errnum IN 20000 .. 20010
LOOP
  IF ORA_IS_SERVERERROR (errnum)
  THEN
    log_user_defined_error (errnum);
  END IF;
END LOOP;
```



Все номера ошибок Oracle отрицательны, кроме 1 (исключение, определяемое пользователем) и 100 (синоним 1403, NO_DATA_FOUND). Однако номер ошибки, передаваемый при вызове `ORA_IS_SERVERERROR`, должен быть положительным, как в приведенном примере.

Центральный обработчик ошибок

Реализовать отдельные триггеры `SERVERERROR` в каждой схеме базы данных возможно, но я рекомендую создать единый центральный триггер с сопроводительным пакетом PL/SQL для предоставления следующих возможностей:

- *Централизованная обработка ошибок* — всего один триггер и пакет, которые хранятся в памяти Oracle (централизация также упрощает сопровождение).
- *Сеансовый журнал ошибок с возможностью поиска* — журнал ошибок может накапливаться во время сеанса. Поиск по журналу позволяет получить такую информацию, как количество вхождений ошибки, временные метки первого и последнего вхождения и т. д. Также возможна очистка журнала по требованию.
- *Возможность сохранения журнала ошибок* — при необходимости журнал можно сохранить в таблице.
- *Возможность просмотра текущего журнала* — текущий журнал ошибок может просматриваться по конкретному номеру ошибки и/или диапазону дат.

Реализация такого централизованного пакета обработки ошибок содержится в файле `error_log.sql` на сайте книги. Когда пакет будет создан, триггер `SERVERERROR` создается так:

```
CREATE OR REPLACE TRIGGER error_log
AFTER SERVERERROR
ON DATABASE
BEGIN
  central_error_log.log_error;
END;
```

Рассмотрим несколько примеров использования. Для начала сгенерируем ошибку:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
*
```

```
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
```

Теперь проведем поиск по номеру ошибки и прочитаем информацию в запись:

```
DECLARE
  v_find_record central_error_log.v_find_record;
BEGIN
  central_error_log.find_error(6502,v_find_record);
  DBMS_OUTPUT.PUT_LINE('Total Found = ' || v_find_record.total_found);
  DBMS_OUTPUT.PUT_LINE('Min Timestamp = ' || v_find_record.min_timestamp);
  DBMS_OUTPUT.PUT_LINE('Max Timestamp = ' || v_find_record.max_timestamp);
END;
```

Результат:

```
Total Found = 1
Min Timestamp = 04-JAN-02
Max Timestamp = 04-JAN-02
```

Триггеры INSTEAD OF

Триггеры INSTEAD OF предназначены для выполнения операций вставки, обновления и удаления элементов *представлений*, но не таблиц. С их помощью можно сделать не-обновляемое представление обновляемым и изменить поведение обновляемого представления по умолчанию.

Создание триггера INSTEAD OF

Команда создания (или замены) триггера INSTEAD OF имеет следующий синтаксис:

```
1 CREATE [OR REPLACE] TRIGGER имя_триггера
2 INSTEAD OF операция
3 ON имя_представления
4 FOR EACH ROW
5 BEGIN
6 ... Код ...
7 END;
```

Основные элементы определения представлены в следующей таблице.

| Строки | Описание |
|--------|--|
| 1 | Создание триггера с заданным именем. Если триггер существует, а предложение REPLACE отсутствует, попытка создания триггера приведет к ошибке ORA-4081 |
| 2 | В этой строке проявляются различия между триггерами INSTEAD OF и всеми остальными типами триггеров. Поскольку триггеры INSTEAD OF запускаются не при наступлении определенных событий, а выполняются вместо команд SQL, для них не нужно указывать ни ключевое слово AFTER или BEFORE, ни имя события. Задается только команда (INSERT, UPDATE, MERGE или DELETE), вместо которой должен выполняться триггер |
| 3 | Строка аналогична соответствующей строке триггеров DDL и событий базы данных, однако вместо ключевого слова DATABASE или SCHEMA в ней задается имя представления, с которым связан триггер |
| 4–7 | Стандартный код PL/SQL |

Принцип действия триггеров INSTEAD OF проще всего объяснить на конкретном примере. Допустим, в системе доставки пиццы используются три таблицы: для учета доставленной продукции, для хранения списка районов доставки и для информации о курьерах:

```
/* Файл в Сети: pizza_tables.sql */
CREATE TABLE delivery
(delivery_id NUMBER,
 delivery_start DATE,
 delivery_end DATE,
 area_id NUMBER,
 driver_id NUMBER);
```

продолжение ➤

```
CREATE TABLE area
  (area_id NUMBER, area_desc  VARCHAR2(30));
```

```
CREATE TABLE driver
  (driver_id NUMBER, driver_name  VARCHAR2(30));
```

Для простоты в этом примере не будут использоваться ни первичные, ни внешние ключи. Нам понадобятся три последовательности, обеспечивающие уникальные идентификаторы для таблиц:

```
CREATE SEQUENCE delivery_id_seq;
CREATE SEQUENCE area_id_seq;
CREATE SEQUENCE driver_id_seq;
```

Чтобы не объяснять работникам фирмы тонкости нормализации и проектирования реляционных баз данных, мы объединим всю информацию о доставках, районах и курьерах в одно представление:

```
VIEW delivery_info AS
SELECT d.delivery_id,
       d.delivery_start,
       d.delivery_end,
       a.area_desc,
       dr.driver_name
FROM   delivery    d,
       area        a,
       driver       dr
WHERE  a.area_id = d.area_id
      AND dr.driver_id = d.driver_id
```

Поскольку вся система запросов строится на основе этого представления, почему бы не использовать его для вставки, обновления и удаления данных? Мы не можем напрямую применять команды DML к этому представлению, так как оно является объединением нескольких таблиц. Как база данных узнает, что следует делать при получении команды INSERT? Необходимо предельно четко объяснить ей, как выполнять операции вставки, обновления и удаления для представления `delivery_info`; иначе говоря, нужно указать, что делать *вместо* обычной вставки, обновления или удаления. В этом нам помогут замещающие триггеры `INSTEAD OF`. Начнем с триггера `INSERT`.

Триггер `INSTEAD OF INSERT`

Триггер `INSERT` должен выполнить четыре основные операции:

- убедиться в том, что столбец `delivery_end` содержит `NULL` (информация о доставках должна вноситься только посредством обновления записи);
- определить идентификатор курьера по заданному имени. Если имя в таблице отсутствует, триггер присваивает курьеру другой идентификатор и добавляет строку в таблицу курьеров, записав в нее заданное имя и новый идентификатор;
- определить идентификатор района по заданному названию. Если имя в таблице отсутствует, триггер присваивает району другой идентификатор и добавляет строку в таблицу районов, записав в нее заданное имя и новый идентификатор;
- добавить строку в таблицу доставки.

Имейте в виду, что этот пример всего лишь демонстрирует вариант использования триггеров, а не эффективные приемы построения бизнес-систем! Через некоторое время в таблицах скопятся повторяющиеся строки с идентификаторами. Однако это представление ускоряет работу, не требуя обязательного предварительного определения списков курьеров и районов.

```

/* Файл в Сети: pizza_triggers.sql */
TRIGGER delivery_info_insert
  INSTEAD OF INSERT
  ON delivery_info
DECLARE
  -- Курсор для получения идентификатора курьера по имени
  CURSOR curs_get_driver_id (cp_driver_name VARCHAR2)
  IS
    SELECT driver_id
      FROM driver
     WHERE driver_name = cp_driver_name;

  v_driver_id  NUMBER;
  -- курсор для получения идентификатора района по названию
  CURSOR curs_get_area_id (cp_area_desc VARCHAR2)
  IS
    SELECT area_id
      FROM area
     WHERE area_desc = cp_area_desc;

  v_area_id    NUMBER;
BEGIN
  /* Значение столбца delivery_end должно быть равно NULL
  */
  IF :NEW.delivery_end IS NOT NULL
  THEN
    raise_application_error
      (-20000
      , 'Delivery end date value must be NULL when delivery created'
      );
  END IF;

  /*
  || Попытка получить идентификатор курьера по имени.
  || Если имя не найдено, создаем новый идентификатор
  || на основе последовательности.
  */
  OPEN curs_get_driver_id (UPPER (:NEW.driver_name));

  FETCH curs_get_driver_id
    INTO v_driver_id;

  IF curs_get_driver_id%NOTFOUND
  THEN
    SELECT driver_id_seq.NEXTVAL
      INTO v_driver_id
      FROM DUAL;

    INSERT INTO driver
      (driver_id, driver_name
      )
      VALUES (v_driver_id, UPPER (:NEW.driver_name)
      );
  END IF;

  CLOSE curs_get_driver_id;

  /*
  || Попытка получить идентификатор района по названию.
  || Если название не найдено, создаем новый идентификатор
  || на основе последовательности.
  */
  OPEN curs_get_area_id (UPPER (:NEW.area_desc));

```

```

FETCH curs_get_area_id
  INTO v_area_id;

IF curs_get_area_id%NOTFOUND
THEN
  SELECT area_id_seq.NEXTVAL
    INTO v_area_id
    FROM DUAL;

  INSERT INTO area
    (area_id, area_desc
     )
    VALUES (v_area_id, UPPER (:NEW.area_desc)
    );
END IF;

CLOSE curs_get_area_id;

/*
|| Добавление строки с информацией о доставке
*/
INSERT INTO delivery
  (delivery_id, delivery_start
   , delivery_end, area_id, driver_id
   )
  VALUES (delivery_id_seq.NEXTVAL, NVL (:NEW.delivery_start, SYSDATE)
    , NULL, v_area_id, v_driver_id
    );
END;
```

Триггер INSTEAD OF UPDATE

Перейдем к триггеру UPDATE. Чтобы упростить код, мы будем обновлять только поле `delivery_end` и только в том случае, если оно содержит значение NULL (нельзя допускать, чтобы курьеры могли изменить время доставки):

```

/* Файл в Сети: pizza_triggers.sql */
TRIGGER delivery_info_update
  INSTEAD OF UPDATE
  ON delivery_info
DECLARE
  -- курсор для получения строки доставки
  CURSOR curs_get_delivery (cp_delivery_id NUMBER)
  IS
    SELECT delivery_end
      FROM delivery
     WHERE delivery_id = cp_delivery_id
    FOR UPDATE OF delivery_end;

  v_delivery_end  DATE;
BEGIN
  OPEN curs_get_delivery (:NEW.delivery_id);
  FETCH curs_get_delivery INTO v_delivery_end;
  IF v_delivery_end IS NOT NULL
  THEN
    RAISE_APPLICATION_ERROR (
      -20000, 'The delivery end date has already been set');
  ELSE
    UPDATE delivery
      SET delivery_end = :NEW.delivery_end
     WHERE CURRENT OF curs_get_delivery;
  END IF;

  CLOSE curs_get_delivery;
END;
```

Триггер INSTEAD OF DELETE

Триггер DELETE в рассматриваемом примере имеет самую простую структуру. Его основная задача — следить за тем, чтобы никто не удалил заполненную строку, а затем самому удалить указанную строку доставки. Строки в таблицах курьеров и районов остаются неизменными.

```
/* Файл в Сети: pizza_triggers.sql */
TRIGGER delivery_info_delete
  INSTEAD OF DELETE
  ON delivery_info
BEGIN
  IF :OLD.delivery_end IS NOT NULL
  THEN
    RAISE_APPLICATION_ERROR (
      -20000, 'Completed deliveries cannot be deleted');
  END IF;

  DELETE delivery
  WHERE delivery_id = :OLD.delivery_id;
END;
```

Заполнение таблиц

Теперь для заполнения всех трех таблиц достаточно одной команды INSERT, которая содержит известную информацию о доставке (курьер и район):

```
SQL> INSERT INTO delivery_info(delivery_id,
2      delivery_start,
3      delivery_end,
4      area_desc,
5      driver_name)
6  VALUES
7  NULL, NULL, NULL, 'LOCAL COLLEGE', 'BIG TED');
```

1 row created.

```
SQL> SELECT * FROM delivery;
DELIVERY_ID DELIVERY_ DELIVERY_  AREA_ID DRIVER_ID
-----
1 13-JAN-02          1          1
```

```
SQL> SELECT * FROM area;
```

```
AREA_ID AREA_DESC
-----
1 LOCAL COLLEGE
```

```
SQL> SELECT * FROM driver;
```

```
DRIVER_ID DRIVER_NAME
-----
1 BIG TED
```

Триггеры INSTEAD OF для вложенных таблиц

В Oracle существует много способов хранения сложных структур данных в виде столбцов таблиц или представлений. На логическом уровне такие решения эффективны, поскольку связь между таблицей или представлением и его столбцами предельно очевидна. С технической точки зрения поддержка даже самых простых операций вроде вставки записи в эти сложные структуры потребует не столь очевидных ухищрений. Одна из сложных ситуаций такого рода решается при помощи особой разновидности триггера INSTEAD OF.

Возьмем следующее представление, объединяющее главы книги со строками главы:

```
VIEW book_chapter_view AS
SELECT chapter_number,
       chapter_title,
       CAST(MULTISET(SELECT *
                     FROM book_line
                     WHERE chapter_number = book_chapter.chapter_number)
            AS book_line_t) lines
FROM book_chapter;
```

Пожалуй, смысл такого представления понять довольно трудно (почему бы просто не объединить таблицы напрямую?), но оно наглядно демонстрирует применение триггеров `INSTEAD OF` для столбцов вложенных таблиц, а также для любых объектов или столбцов-коллекций в представлениях.

После создания записи в таблице `BOOK_CHAPTER` и запроса к представлению мы видим, что глава еще не содержит строк:

```
CHAPTER_NUMBER CHAPTER_TITLE
-----
LINES(CHAPTER_NUMBER, LINE_NUMBER, LINE_TEXT)
-----
18 Triggers
BOOK_LINE_T()
```

Соответственно мы пытаемся создать первую строку:

```
SQL> INSERT INTO TABLE(SELECT lines
2          FROM book_chapter_view
3          WHERE chapter_number = 18)
4  VALUES(18,1,'Triggers are...');
INSERT INTO TABLE(SELECT lines
*
```

```
ERROR at line 1:
ORA-25015: cannot perform DML on this nested table view column
```

База данных определила, что для простой вставки значений в таблицу `BOOK_LINE`, замаскированную под столбец `LINES` представления, не хватает информации. Проблема решается с помощью триггера `INSTEAD OF`:

```
TRIGGER lines_ins
INSTEAD OF INSERT ON NESTED TABLE lines OF book_chapter_view
BEGIN
  INSERT INTO book_line
    (chapter_number,
     line_number,
     line_text)
  VALUES(:PARENT.chapter_number,
         :NEW.line_number,
         :NEW.line_text);
END;
```

Теперь мы можем добавить первую строку:

```
SQL> INSERT INTO TABLE ( SELECT lines
2          FROM book_chapter_view
3          WHERE chapter_number = 18 )
4  VALUES(18,1,'Triggers Are...');
```

```
1 row created.
```

```
SQL> SELECT *
2  FROM book_chapter_view;
```



```
CHAPTER_NUMBER CHAPTER_TITLE
```

```
-----  
LINES(CHAPTER_NUMBER, LINE_NUMBER, LINE_TEXT)
```

```
-----  
18 Triggers  
BOOK_LINE_T(BOOK_LINE_O(18, 1, 'Triggers Are...'))
```

Код SQL, используемый для создания этих триггеров, очень похож на аналогичный код для других триггеров **INSTEAD OF**, если не считать пары различий:

- Для обозначения столбца используется предложение **ON NESTED TABLE COLUMN OF**.
- Новая псевдозапись **PARENT** содержит значения записи родительского представления.

Триггеры AFTER SUSPEND

В Oracle9i Release 1 появился новый тип триггеров, запускаемых в ответ на приостановление выполняемых команд. Приостановление может быть вызвано разными причинами — например, превышением квоты выделенного процессу табличного пространства. Триггер **AFTER SUSPEND** должен решить проблему, чтобы выполнение команды было возобновлено. Это просто подарок для разработчиков, которым надоели проблемы с нехваткой памяти, и для администраторов базы данных, которым приходится эти проблемы решать.

Синтаксис создания триггера **AFTER SUSPEND** сходен с синтаксисом создания триггеров **DDL** и триггеров событий базы данных. В нем объявляется инициирующее событие (**SUSPEND**), время запуска (**AFTER**) и область действия (**DATABASE** или **SCHEMA**):

```
1 CREATE [OR REPLACE] TRIGGER имя_триггера  
2 AFTER SUSPEND  
3 ON {DATABASE | SCHEMA}  
4 BEGIN  
5 ... код ...  
6 END;
```

В знакомстве с **AFTER SUSPEND** нам поможет пример типичной ситуации, требующей создания этого вида триггеров.

Допустим, некоему разработчику приходится сопровождать сотни программ, которые выполняются в течение ночи, выполняют продолжительные транзакции для обобщения информации и ее перемещения между разными приложениями. По крайней мере дважды в неделю разработчик получает ночью сообщения на пейджер, потому что в одной из его программ возникла ошибка Oracle:

```
ERROR at line 1:  
ORA-01536: space quota exceeded for tablespace 'USERS'
```

Следующий неизбежный шаг — звонить администратору базы данных и просить увеличить пространственную квоту. Администратор спрашивает: «А сколько нужно?», на что разработчик может ответить лишь неубедительным: «Даже не знаю, объем данных сильно изменяется». Оба остаются весьма недовольными, потому что администратору нужен контроль за выделяемым пространством в целях планирования, а разработчику не хочется так часто просыпаться от служебных сообщений.

Настройка для триггера AFTER SUSPEND

К счастью, триггер **AFTER SUSPEND** может решить проблемы обоих героев. Посмотрим, как же решается эта задача. Разработчик обнаруживает конкретную точку в коде, в которой чаще всего встречается ошибка. Это внешне непримечательная команда **INSERT** в конце программы, выполняющейся несколько часов:

```
INSERT INTO monthly_summary (  
    acct_no, trx_count, total_in, total_out)  
VALUES (  
    v_acct, v_trx_count, v_total_in, v_total_out);
```

Больше всего бесит то, что на вычисления уходит несколько часов, а вычисленные значения немедленно теряются при сбое завершающей команды `INSERT`. По меньшей мере разработчик хочет, чтобы программа приостанавливалась, пока он обращается к администратору за дополнительным пространством. Он обнаруживает, что это можно сделать простой командой `ALTER SESSION`:

```
ALTER SESSION ENABLE RESUMABLE TIMEOUT 3600 NAME 'Monthly Summary';
```

Это означает, что каждый раз, когда сеанс Oracle обнаруживает ошибку нехватки пространства, он переходит в приостановленное состояние на 3600 секунд (1 час) — если повезет, то с возможностью продолжения. Это дает необходимое время, чтобы система мониторинга могла отправить сообщение, разработчик обратился к администратору, а тот выделил дополнительное пространство. Система не идеальная, но по крайней мере часы, потраченные за вычисления, не будут теряться понапрасну.

Другая проблема заключается в том, что при попытках разрешить ситуацию в середине ночи и разработчик и администратор чувствуют такую усталость и раздражительность, что на разрешение недоразумений уходит лишнее время. На помощь приходит другая особенность приостановленных команд: представление `DBA_RESUMABLE`. В нем содержится информация обо всех сеансах, зарегистрированных для продолжения приведенной командой `ALTER SESSION`.



Чтобы пользователи могли включить режим приостановки, им должна быть предоставлена системная привилегия `RESUMABLE`.

НЕДОПУСТИМЫЕ ОПЕРАЦИИ DDL В СИСТЕМНЫХ ТРИГГЕРАХ

Триггерам `AFTER SUSPEND` не разрешается выполнять операции DDL (`ALTER USER` и `ALTER TABLESPACE`) для решения диагностируемых проблем. В таких случаях они просто инициируют ошибку `ORA-30511` (недопустимая операция DDL в системных триггерах). Один из способов решения проблемы заключается в следующем:

1. В триггере `AFTER SUSPEND` запишите в таблицу команду SQL, необходимую для решения проблемы.
2. Создайте пакет PL/SQL, который читает команды SQL из таблицы и выполняет их.
3. Организуйте ежеминутную передачу пакета PL/SQL пакету `DBMS_JOB`.

Теперь когда программа переходит в приостановленное состояние, разработчику остается лишь позвонить администратору и пробормотать в трубку: «Проверь представление `dba_resumable`». Администратор запрашивает информацию со своей учетной записи и смотрит, что же происходит:

```
SQL> SELECT session_id,  
2         name,  
3         status,  
4         error_number  
5 FROM dba_resumable
```

| SESSION_ID NAME | STATUS | ERROR_NUMBER |
|-------------------|-----------|--------------|
| ----- | ----- | ----- |
| 8 Monthly Summary | SUSPENDED | 1536 |

1 row selected.

Из результатов видно, что сеанс 8 приостановлен из-за ошибки ORA-01536 (превышение пространственной квоты). По прошлому опыту администратор знает, какая схема и табличное пространство виноваты в случившемся, он устраняет проблему и бормочет в трубку: «Готово». Приостановленная команда в коде разработчика немедленно возобновляет выполнение, а разработчик и администратор могут спать спокойно.

Код триггера

Через несколько недель и разработчику, и администратору надоедают повторяющиеся (хотя и с меньшей частотой) ночные переговоры, и администратор решает автоматизировать свою работу при помощи триггера AFTER SUSPEND. Фрагмент кода, который он пишет и устанавливает под административной учетной записью:

```
/* Файл в Сети: smart_space_quota.sql */
TRIGGER after_suspend
AFTER SUSPEND
ON DATABASE
DECLARE
...
BEGIN

-- Если это ошибка, связанная с доступным пространством...
IF ORA_SPACE_ERROR_INFO ( error_type => v_error_type,
                           object_type => v_object_type,
                           object_owner => v_object_owner,
                           table_space_name => v_tbspc_name,
                           object_name => v_object_name,
                           sub_object_name => v_subobject_name ) THEN

-- Если произошло превышение квоты табличного пространства...
IF v_error_type = 'SPACE QUOTA EXCEEDED' AND
   v_object_type = 'TABLE SPACE' THEN

-- Получение имени пользователя
OPEN curs_get_username;
FETCH curs_get_username INTO v_username;
CLOSE curs_get_username;

-- Получение текущей квоты для имени пользователя и табличного пространства
OPEN curs_get_ts_quota(v_object_name,v_username);
FETCH curs_get_ts_quota INTO v_old_quota;
CLOSE curs_get_ts_quota;

-- Создание команды ALTER USER и ее отправка заданию
-- (потому что при попытке выполнить ее "на месте"
-- произойдет ошибка ORA-30511)
v_new_quota := v_old_quota + 40960;
v_sql := 'ALTER USER ' || v_username || ' ' ||
         'QUOTA ' || v_new_quota || ' ' ||
         'ON ' || v_object_name;
fixer.fix_this(v_sql);

END IF; -- Превышение квоты табличного пространства

END IF; -- Ошибка, связанная с пространством

END;
```

Этот фрагмент создает триггер, который срабатывает каждый раз, когда команда переходит в приостановленное состояние, и пытается решить проблему. (Обратите внимание: в этом конкретном примере исправляются только ошибки превышения квот табличного пространства.)

Теперь при возникновении проблем с квотами табличного пространства триггер AFTER SUSPEND уровня базы данных срабатывает и помещает команду SQL в таблицу при помощи служебного пакета `fixer`. Задание `fixer`, работающее в фоновом режиме, выбирает команду SQL из таблицы и выполняет ее, решая проблему с квотой без лишних телефонных звонков.



Полный код триггера AFTER SUSPEND и пакета `fixer` содержится в файле `fixer.sql` на сайте книги.

Функция ORA_SPACE_ERROR_INFO

Информацию о причине приостановки команды можно получить при помощи функции `ORA_SPACE_ERROR_INFO`, представленной в предшествующих примерах. Рассмотрим синтаксис вызова этой функции; ее параметры перечислены в табл. 19.3.

Таблица 19.3. Параметры `ORA_SPACE_ERROR_INFO`

| Строки | Описание |
|-----------------------------|---|
| тип_ошибки | Тип ошибки, связанной с пространственной ошибкой; одно из следующих значений: – SPACE QUOTA EXCEEDED: пользователь превысил свою квоту табличного пространства – MAX EXTENTS REACHED: объект пытается выйти за пределы заданного максимального количества сегментов – NO MORE SPACE: в табличном пространстве не хватает места для сохранения новой информации |
| тип_объекта | Тип объекта, для которого произошла пространственная ошибка |
| владелец_объекта | Владелец объекта, для которого произошла пространственная ошибка |
| имя_табличного_пространства | Табличное пространство, для которого произошла пространственная ошибка |
| имя_объекта | Имя объекта, для которого произошла пространственная ошибка |
| имя_подобъекта | Имя подобъекта, для которого произошла пространственная ошибка |

Функция возвращает логическое значение `TRUE`, если приостановка происходит из-за одной из ошибок, перечисленных в таблице, или `FALSE` в противном случае.

Функция `ORA_SPACE_ERROR_INFO` не исправляет проблемы с пространством в вашей системе; она всего лишь предоставляет информацию, необходимую для выполнения дальнейших действий. Вы уже видели, как решалась проблема с квотой в предыдущем примере. Ниже приводятся два дополнительных примера SQL, которые могут использоваться для решения проблем с пространством, выявляемых функцией `ORA_SPACE_ERROR_INFO`:

- Таблица (или индекс) достигла максимального количества сегментов, и дополнительные сегменты недоступны:

```
ALTER тип_объекта владелец_объекта.имя_объекта STORAGE (MAXEXTENTS UNLIMITED);
```

- В табличном пространстве полностью закончилось место:

```

/* Предполагается использование Oracle Managed Files
   (Oracle9i Database и выше), так что явное объявление файла данных
   не обязательно */
ALTER TABLESPACE имя_табличного_пространства ADD DATAFILE;

```

Пакет DBMS_RESUMABLE

Если функция `ORA_SPACE_ERROR_INFO` возвращает `FALSE`, значит, ситуация, вызвавшая приостановку команды, не может быть исправлена, а следовательно, разумных причин оставаться в приостановленном состоянии тоже нет. Такие команды можно прервать из триггера `AFTER_SUSPEND` при помощи процедуры `ABORT` из пакета `DBMS_RESUMABLE`. Пример:

```

/* Файл в Сети: local_abort.sql */
TRIGGER after_suspend
AFTER SUSPEND
ON SCHEMA
DECLARE
    CURSOR curs_get_sid IS
    SELECT sid
    FROM v$session
    WHERE audsid = SYS_CONTEXT('USERENV','SESSIONID');
    v_sid          NUMBER;
    v_error_type   VARCHAR2(30);
    ...

BEGIN

    IF ORA_SPACE_ERROR_INFO(...
        ...Пытаемся исправить...
    ELSE -- can't fix the situation
        OPEN curs_get_sid;
        FETCH curs_get_sid INTO v_sid;
        CLOSE curs_get_sid;
        DBMS_RESUMABLE.ABORT(v_sid);
    END IF;

END;

```

Процедура `ABORT` получает один аргумент: идентификатор прерываемого сеанса. Это позволяет вызывать `ABORT` из триггеров `AFTER SUSPEND` из уровня `DATABASE` или `SCHEMA`. Прерванный сеанс получает ошибку `ORA-01013` (пользователь запросил операцию отмены текущей операции).

Действительно, отмена была запрошена пользователем — но каким именно? Кроме процедуры `ABORT`, пакет `DBMS_RESUMABLE` содержит функции и процедуры для чтения и назначения тайм-аута:

- `GET_SESSION_TIMEOUT` — возвращает значение тайм-аута приостановленного сеанса по идентификатору сеанса:

```

FUNCTION DBMS_RESUMABLE.GET_SESSION_TIMEOUT (sessionid IN NUMBER)
RETURN NUMBER;

```

- `SET_SESSION_TIMEOUT` — задает значение тайм-аута приостановленного сеанса по идентификатору сеанса:

```

PROCEDURE DBMS_RESUMABLE.SET_SESSION_TIMEOUT (sessionid IN NUMBER,
    TIMEOUT IN NUMBER);

```

- `GET_TIMEOUT` — возвращает значение тайм-аута текущего сеанса:

```

FUNCTION DBMS_RESUMABLE.GET_TIMEOUT RETURN NUMBER;

```

- `SET_TIMEOUT` — задает значение тайм-аута текущего сеанса:

```

PROCEDURE DBMS_REUSABLE.SET_TIMEOUT (TIMEOUT IN NUMBER);

```



Новые значения тайм-аута вступают в действие немедленно, но не обнуляют счетчик.

Многократное срабатывание

Триггеры AFTER SUSPEND срабатывают при каждой приостановке команды. Следовательно, они могут многократно сработать во время выполнения одной команды. Например, представьте, что в системе реализован следующий жестко запрограммированный триггер:

```
/* Файл в Сети: increment_extents.sql */
TRIGGER after_suspend
AFTER SUSPEND ON SCHEMA
DECLARE
    -- Получение нового максимума (текущий плюс 1)
    CURSOR curs_get_extents IS
    SELECT max_extents + 1
        FROM user_tables
        WHERE table_name = 'MONTHLY_SUMMARY';
    v_new_max NUMBER;

BEGIN
    -- Выборка нового максимального количества сегментов
    OPEN curs_get_extents;
    FETCH curs_get_extents INTO v_new_max;
    CLOSE curs_get_extents;
    -- alter the table to take on the new value for maxextents
    EXECUTE IMMEDIATE 'ALTER TABLE MONTHLY_SUMMARY ' ||
        'STORAGE ( MAXEXTENTS ' ||
        v_new_max || ')';
    DBMS_OUTPUT.PUT_LINE('Incremented MAXEXTENTS to ' || v_new_max);
END;
```

Если начать с пустой таблицы со значением MAXEXTENTS (максимальное количество сегментов), равным 1, вставка данных объемом 4 сегмента дает следующий результат:

```
SQL> @test
```

```
Incremented MAXEXTENTS to 2
Incremented MAXEXTENTS to 3
Incremented MAXEXTENTS to 4
```

```
PL/SQL procedure successfully completed.
```

Исправлять или не исправлять?

Вот в чем вопрос! Предшествующие примеры показали, как ошибки нехватки места исправляются на ходу — команды приостанавливаются до того, как вмешательство (ручное или автоматизированное) позволит им продолжить работу. Если довести происходящее до крайности, этот метод позволяет устанавливать приложения с минимальными требованиями к табличному пространству, квотам и сегментам, а затем наращивать их по мере надобности. Слишком старательному администратору базы данных такая ситуация покажется идеальной, но у нее есть свои недостатки:

- *Нерегулярные паузы* — паузы с приостановкой команд могут породить хаос в крупномасштабных оперативных системах обработки транзакций (OLTP), требующих высокой пропускной способности. Ситуация дополнительно усугубляется, если на исправление уходит много времени.

- *Конкуренция за ресурсы* — приостановленные команды удерживают блокировку таблиц. Это может привести к простоям или сбоям при выполнении других команд.
- *Непроизводительные затраты* — ресурсы, необходимые для частого добавления сегментов или файлов данных или увеличения квот, могут оказаться неприемлемо высокими по сравнению с ресурсами, необходимыми для непосредственной работы приложения.

По этим причинам я рекомендую осторожно использовать триггеры AFTER SUSPEND. Они идеально подходят для долгих процессов, которые должны перезапускаться после сбоя, а также для пошаговых процессов, для перезапуска которых необходима отмена внесенных изменений командами DML. В приложениях OLTP они работают недостаточно хорошо.

Сопровождение триггеров

Oracle поддерживает ряд команд DDL, повышающих эффективность управления триггерами. С их помощью можно включать, отключать и удалять триггеры, просматривать информацию о них, проверять их состояние.

Отключение, включение и удаление триггеров

Отключенный триггер не запускается при наступлении связанного с ним события. Удаленный триггер полностью исчезает из базы данных. Синтаксис отключения триггера очень прост по сравнению с синтаксисом создания:

```
ALTER TRIGGER имя_триггера DISABLE;
```

Пример:

```
ALTER TRIGGER emp_after_insert DISABLE;
```

Отключенный триггер можно включить повторно следующей командой:

```
ALTER TRIGGER emp_after_insert ENABLE;
```

В команде ALTER TRIGGER задается только имя триггера; ни его тип, ни что-либо иное задавать не нужно. Эту команду можно вызывать из хранимой процедуры, как в следующем примере, где включение и отключение всех триггеров таблицы выполняется динамическим кодом SQL:

```
/* Файл в Сети: settrig.sp */
PROCEDURE settrig (
    tab      IN   VARCHAR2
    , sch     IN   VARCHAR DEFAULT NULL
    , action  IN   VARCHAR2
)
IS
    l_action      VARCHAR2 (10) := UPPER (action);
    l_other_action VARCHAR2 (10) := 'DISABLED';
BEGIN
    IF l_action = 'DISABLE'
    THEN
        l_other_action := 'ENABLED';
    END IF;

    FOR rec IN (SELECT trigger_name FROM user_triggers
                WHERE table_owner = UPPER (NVL (sch, USER))
                  AND table_name = tab AND status = l_other_action)
    LOOP
        EXECUTE IMMEDIATE
            'ALTER TRIGGER ' || rec.trigger_name || ' ' || l_action;
    END LOOP;
END;
```

Команда удаления триггера `DROP TRIGGER` столь же проста; как и в предыдущем случае, достаточно указать имя:

```
DROP TRIGGER emp_after_insert;
```

Создание отключенных триггеров

В Oracle11g появилась возможность создания триггеров в отключенном состоянии. Например, это может быть удобно, если вы хотите проверить правильность триггера без его запуска. Очень простой пример:

```
TRIGGER just_testing
AFTER INSERT ON abc
DISABLE
BEGIN
    NULL;
END;
```

Благодаря присутствию в заголовке ключевого слова `DISABLE` этот триггер будет проверен, откомпилирован и создан, но не будет запускаться до его явного включения на более поздней стадии. Учтите, что ключевое слово `DISABLE` не сохраняется в базе данных:

```
SQL> SELECT trigger_body
      2 FROM user_triggers
      3 WHERE trigger_name = 'JUST_TESTING';
```

```
TRIGGER_BODY
-----
BEGIN
    NULL;
END;
```

Если вы пользуетесь служебными программами с графическим интерфейсом, будьте осторожны — перекомпиляция может привести к непреднамеренному включению триггеров.

Просмотр триггеров

В словаре данных Oracle имеется несколько представлений, возвращающих разнообразную информацию о триггерах:

- `DBA_TRIGGERS` — все триггеры базы данных;
- `ALL_TRIGGERS` — все триггеры, доступные текущему пользователю;
- `USER_TRIGGERS` — все триггеры, принадлежащие текущему пользователю.

В табл. 19.4 перечислены самые важные (и часто используемые) столбцы этих представлений.

Таблица 19.4. Название таблицы

| Имя | Описание |
|------------------|---|
| TRIGGER_NAME | Имя триггера |
| TRIGGER_TYPE | Тип триггера: для триггеров DML—BEFORE_STATEMENT, BEFORE EACH ROW, AFTER EACH ROW или AFTER STATEMENT; для триггеров DDL — BEFORE EVENT или AFTER EVENT; для триггеров INSTEAD OF — INSTEAD OF; для триггеров AFTER_SUSPEND — AFTER EVENT |
| TRIGGERING_EVENT | Событие, вызвавшее запуск триггера: для триггеров DML — UPDATE, INSERT или DELETE; для триггеров DDL — операция DDL (см. список в разделе, посвященном триггерам данного типа); для триггеров событий базы данных — ERROR, LOGON, LOGOFF, STARTUP или SHUTDOWN; для триггеров INSTEAD OF — INSERT, UPDATE или DELETE; для триггеров AFTER SUSPEND — SUSPEND |

| Имя | Описание |
|-------------------|--|
| TABLE_OWNER | Различная информация в зависимости от типа триггера: для триггеров DML — имя владельца таблицы, с которой связан триггер; для триггеров DDL при условии, что это триггер уровня базы данных, — SYS, в противном случае — имя владельца триггера; для триггеров событий базы данных при условии, что это триггеры уровня базы данных, — SYS, в противном случае — имя владельца триггера; для триггеров INSTEAD OF — имя владельца представления, с которым связан данный триггер; для триггеров AFTER SUSPEND при условии, что это триггеры уровня базы данных, — SYS, в противном случае — имя владельца триггера |
| BASE_OBJECT_TYPE | Тип объекта, с которым связан триггер: для триггеров DML — TABLE; для триггеров DDL — SCHEMA или DATABASE; для триггеров событий базы данных — SCHEMA или DATABASE; для триггеров INSTEAD OF — VIEW; для триггеров AFTER SUSPEND — SCHEMA или DATABASE |
| TABLE_NAME | Для триггеров DML — имя таблицы, с которой связан триггер; для остальных типов триггеров — значение NULL |
| REFERENCING_NAMES | Для триггеров DML (уровня строки) — предложение, используемое для определения псевдонимов записей OLD и NEW; для остальных типов триггеров — текст «REFERENCING NEW AS NEW OLD AS OLD» |
| WHEN_CLAUSE | Для триггеров DML — предложение с условием выполнения триггера |
| STATUS | Состояние триггера (ENABLED или DISABLED) |
| ACTION_TYPE | Признак того, выполняет ли триггер вызов (CALL) или содержит код PL/SQL (PL/SQL) |
| TRIGGER_BODY | Текст тела триггера (столбец типа LONG); начиная с Oracle9i эта информация также присутствует в таблице USER_SOURCE |

Проверка работоспособности триггера

Как ни странно, ни одно из указанных представлений не содержит информации о том, находится ли триггер в работоспособном состоянии. Если триггер создан ошибочной командой PL/SQL, он сохраняется в базе данных, но помечается как неработоспособный (INVALID). Чтобы определить, находится ли триггер в этом состоянии, нужно извлечь информацию из представления USER_OBJECTS или ALL_OBJECTS:

```
SQL> CREATE OR REPLACE TRIGGER invalid_trigger
2  AFTER DDL ON SCHEMA
3  BEGIN
4  NULL
5  END;
6  /
```

Warning: Trigger created with compilation errors.

```
SQL> SELECT object_name,
2         object_type,
3         status
4  FROM user_objects
5  WHERE object_name = 'INVALID_TRIGGER';
```

```
OBJECT_NAME    OBJECT TYPE    STATUS
-----
INVALID_TRIGGER TRIGGER        INVALID
```

20

Управление приложениями PL/SQL

Написание программного кода — всего лишь одна из составляющих длительного процесса разработки и сопровождения приложений. Полностью описать весь жизненный цикл проектирования, разработки и развертывания в одной главе невозможно; мы только предлагаем ряд идей и советов по следующим вопросам:

- **Управление программным кодом и его анализ в базе данных.** При компиляции программ PL/SQL исходный код загружается в словарь базы данных в разных формах (в виде текста программы, зависимостей, информации о параметрах и т. д.). С помощью SQL из этого словаря можно запрашивать информацию, необходимую для управления программным кодом.
- **Управление зависимостями и перекомпиляция.** Oracle автоматически управляет зависимостями между объектами базы данных. Очень важно понимать, как работают эти зависимости, как свести к минимуму последствия изменений и как лучше всего перекомпилировать программные модули.
- **Тестирование программ PL/SQL.** Тестирование программ с целью проверки их правильности играет важнейшую роль в успешной разработке и развертывании приложений. Самостоятельно разработанные тесты можно усилить при помощи инфраструктур автоматизированного тестирования — как коммерческих, так и расширяемых с открытым кодом.
- **Трассировка кода PL/SQL.** Многие современные приложения очень сложны — настолько, что разработчики нередко начинают путаться в собственном коде. Трассировочные вызовы в ваших программах могут принести дополнительную информацию, которая поможет вам разобраться в программе.
- **Отладка программ PL/SQL.** В состав многих средств разработки сейчас входят графические отладчики на основе API DBMS_DEBUG. Это очень мощные инструменты, но все же они охватывают лишь малую часть всего процесса отладки.
- **Защита хранимого кода.** Oracle предоставляет такой способ хранения исходного кода, при котором конфиденциальная информация скрывается от посторонних глаз. Эта технология полезна для производителей коммерческих приложений на базе хранимого кода PL/SQL.
- **Оперативная замена.** Эта функция, появившаяся в Oracle11g Release 2, позволяет администраторам баз данных на ходу модифицировать код приложений PL/SQL. Новые версии кода и таблицы баз данных компилируются и подключаются к приложению во время его использования, что сокращает время неработоспособности. Хотя эта тема скорее относится к области деятельности администраторов баз данных, она кратко описывается в этой главе.

Управление программным кодом в базе данных

При компиляции программного модуля PL/SQL его исходный код сохраняется в базе данных. Это дает разработчикам два важных преимущества:

Информацию о программном коде можно получить с помощью запросов SQL. Разработчики могут писать запросы и даже целые программы PL/SQL, которые читают информацию из представлений словаря данных и даже могут изменять состояние кода приложения.

База данных управляет зависимостями между хранимыми объектами. В мире PL/SQL не существует процесса «компоновки» исполняемых файлов, которые затем запускаются пользователями. База данных берет на себя все служебные операции, позволяя разработчику сосредоточиться на реализации бизнес-логики.

В следующих разделах представлены основные источники информации в словаре данных.

Представления словаря данных

Словарь данных Oracle — настоящие джунгли! Он изобилует полезной информацией, но найти путь к цели порой бывает очень непросто. В нем сотни представлений, основанных на сотнях таблиц, множество сложных взаимосвязей, специальных кодов и слишком много неоптимизированных определений представлений. Вскоре мы рассмотрим подмножество важнейших представлений, а пока выделим три типа (или уровня) представлений словаря данных:

- **USER_*** — представления с информацией об объектах базы данных, принадлежащих текущей схеме.
- **ALL_*** — представления с информацией об объектах базы данных, доступных в текущей схеме (либо принадлежащих ей, либо доступных благодаря соответствующим привилегиям). Обычно они содержат те же столбцы, что и соответствующие представления **USER**, с добавлением столбца **OWNER** в представлениях **ALL**.
- **DBA_*** — представления с информацией обо всех объектах базы данных. Обычно содержат те же столбцы, что и соответствующие представления **ALL**. Исключение составляют представления **v\$, gx\$** и **x\$**.

В этой главе мы будем работать с представлениями **USER**; вы можете легко изменить любые сценарии и приемы, чтобы они работали с представлениями **ALL**, добавив в свою логику столбец **OWNER**. Вероятно, для разработчика PL/SQL самыми полезными будут следующие представления:

- **USER_ARGUMENTS** — аргументы (параметры) всех процедур и функций схемы.
- **USER_DEPENDENCIES** — все зависимости (входящие и исходящие) для принадлежащих текущей схеме объектов. Представление в основном используется Oracle для помечки неработоспособных объектов, а также средой разработки для вывода информации зависимостей в программах просмотра объектов. Примечание: для полного анализа зависимостей следует использовать представление **ALL_DEPENDENCIES** на случай, если программная единица будет вызвана другой программной единицей, принадлежащей другой схеме.
- **USER_ERRORS** — текущий набор ошибок компиляции для всех хранимых объектов (включая триггеры). Представление используется командой **SQL*Plus SHOW ERRORS**, описанной в главе 2. Вы также можете писать собственные запросы к этому представлению.
- **USER_IDENTIFIERS** — представление появилось в Oracle11g, а для его заполнения используется утилита PL/Scope. Содержит информацию обо всех идентификаторах

(имена программ, переменных и т. д.) в кодовой базе; исключительно полезный инструмент анализа кода.

- **USER_OBJECTS** — объекты, принадлежащие текущей схеме. Например, при помощи этого представления можно узнать, помечен ли объект как неработоспособный (**INVALID**), найти все пакеты, в имени которых присутствует **EMP**, и т. д.
- **USER_OBJECT_SIZE** — размер принадлежащих текущей схеме объектов. Точнее, это представление возвращает информацию о размере исходного, разобранного и откомпилированного кода. И хотя оно в основном используется компилятором и ядром времени выполнения, вы можете воспользоваться им для поиска больших программ в вашей среде — кандидатов для размещения в **SGA**.
- **USER_PLSQL_OBJECT_SETTINGS** — представление появилось в Oracle10g. Содержит информацию о характеристиках объектов PL/SQL, которые могут изменяться командами **DDL ALTER** и **SET**: уровни оптимизации, параметры отладки и т. д.
- **USER_PROCEDURES** — информация о хранимых программах (*не только* процедурах, как можно было бы подумать по названию) — например, модель **AUTHID**, признак детерминированности функций и т. д.).
- **USER_SOURCE** — исходный код всех принадлежащих текущей схеме объектов (в Oracle9i и выше — вместе с триггерами баз данных и исходным кодом Java). Это очень удобное представление — вы можете применять к нему всевозможные средства анализа исходного кода, используя **SQL** и особенно Oracle Text.
- **USER_STORED_SETTINGS** — флаги компилятора PL/SQL. Это представление поможет узнать, какие программы были откомпилированы в код аппаратной платформы.
- **USER_TRIGGERS** и **USER_TRIG_COLUMNS** — триггеры базы данных, принадлежащие текущей схеме (включая исходный код и описание иницилирующих событий), а также столбцы, связанные с триггерами.

Для просмотра структуры любого из этих представлений можно либо воспользоваться командой **DESCRIBE** в **SQL*Plus**, либо обратиться к документации Oracle. Примеры использования этих представлений приведены в следующем разделе.

Вывод информации о хранимых объектах

В представлении **USER_OBJECTS** содержится ключевая информация об объекте:

- **OBJECT_NAME** — имя объекта.
- **OBJECT_TYPE** — тип объекта (**PACKAGE**, **FUNCTION**, **TRIGGER** и т. д.).
- **STATUS** — состояние объекта (**VALID** или **INVALID**).
- **LAST_DDL_TIME** — время последнего изменения объекта.

Следующий сценарий **SQL*Plus** выводит информацию о состоянии объектов PL/SQL:

```
/* Файл в Сети: psobj.sql */
SELECT object_type, object_name, status
  FROM user_objects
 WHERE object_type IN (
    'PACKAGE', 'PACKAGE BODY', 'FUNCTION', 'PROCEDURE',
    'TYPE', 'TYPE BODY', 'TRIGGER')
 ORDER BY object_type, status, object_name
```

Результат работы сценария выглядит примерно так:

| OBJECT_TYPE | OBJECT_NAME | STATUS |
|-------------|-------------------|---------|
| FUNCTION | DEVELOP_ANALYSIS | INVALID |
| | NUMBER_OF_ATOMICS | INVALID |
| PACKAGE | CONFIG_PKG | VALID |
| | EXCHDLR_PKG | VALID |

Обратите внимание на два модуля с пометкой `INVALID`. О том, как вернуть программный модуль в действительное состояние `VALID`, будет рассказано далее.

Вывод и поиск исходного кода

Исходный код программ следует всегда хранить в текстовых файлах (или в средах разработки, предназначенных для хранения и работы с кодом PL/SQL за пределами Oracle). Однако хранение программ в базе данных позволяет использовать SQL-запросы для анализа исходного кода по всем модулям, что непросто сделать в текстовом редакторе.

Представление `USER_SOURCE` содержит исходный код всех объектов, принадлежащих текущему пользователю. Его структура такова:

| Name | Null? | Type |
|-------|----------|----------------|
| ----- | ----- | ----- |
| NAME | NOT NULL | VARCHAR2(30) |
| TYPE | | VARCHAR2(12) |
| LINE | NOT NULL | NUMBER |
| TEXT | | VARCHAR2(4000) |

Здесь `NAME` — имя объекта, `TYPE` — его тип (от программ PL/SQL до блоков Java и исходного кода триггеров), `LINE` — номер строки, а `TEXT` — текст исходного кода.

Представление `USER_SOURCE` является чрезвычайно ценным источником информации для разработчиков. При помощи соответствующих запросов разработчик может:

- вывести строку исходного кода с заданным номером;
- проверить стандарты кодирования;
- выявить возможные ошибки и дефекты в исходном коде;
- найти программные конструкции, не выявляемые из других представлений.

Предположим, в проекте действует правило, согласно которому отдельные разработчики никогда не должны жестко кодировать пользовательские номера ошибок из диапазона от `-20 999` до `-20 000` (поскольку это может привести к возникновению конфликтов). Конечно, руководитель проекта не может помешать разработчику написать следующую строку:

```
RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
```

Но зато он может создать пакет, который находит все программы с подобными фрагментами. Это очень простой пакет, а его главная процедура имеет вид:

```
/* Файлы в Сети: valstd.* */
PROCEDURE progwith (str IN VARCHAR2)
IS
    TYPE info_rt IS RECORD (
        NAME      user_source.NAME%TYPE
        , text     user_source.text%TYPE
    );
    TYPE info_aat IS TABLE OF info_rt
        INDEX BY PLS_INTEGER;

    info_aa  info_aat;
BEGIN
    SELECT NAME || '-' || line
        , text
    BULK COLLECT INTO info_aa
    FROM user_source
    WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
        AND NAME <> 'VALSTD'
        AND NAME <> 'ERRNUMS';
```

продолжение ➤

```

disp_header ('Checking for presence of ' || str || '');

FOR indx IN info_aa.FIRST .. info_aa.LAST
LOOP
    pl (info_aa (indx).NAME, info_aa (indx).text);
END LOOP;
END progwith;

```

После компиляции этого пакета в схеме можно проверить присутствие в программах значений –20NNN:

```

SQL> EXEC valstd.progwith ('-20')
=====
VALIDATE STANDARDS
=====
Checking for presence of "-20"
CHECK_BALANCE - RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
MY_SESSION - PRAGMA EXCEPTION_INIT(dblink_not_open,-2081);
VSESSTAT - CREATE DATE : 1999-07-20

```

Обратите внимание: третья строка не противоречит правилам; она выводится только потому, что условие отбора сформулировано недостаточно жестко.

Конечно, этот аналитический инструмент весьма примитивен, но при желании его можно усовершенствовать. Можно, к примеру, генерировать HTML-документ с информацией, размещая его в интрасети, или выполнять сценарии `valstd` каждое воскресенье средствами `DBMS_JOB`, чтобы в понедельник утром администратор мог проверить наличие данных обратной связи в интрасети.

Проверка ограничений размера

Представление `USER_OBJECT_SIZE` предоставляет информацию о размере программ, хранимых в базе данных:

- `SOURCE_SIZE` — размер исходного кода в байтах. Код должен находиться в памяти во время компиляции (включая динамическую/автоматическую перекомпиляцию).
- `PARSED_SIZE` — размер объекта в байтах после разбора. Эта форма должна находиться в памяти при компиляции любого объекта, ссылающегося на данный объект.
- `CODE_SIZE` — размер кода в байтах. Код должен находиться в памяти при выполнении объекта.

Следующий запрос выводит кодовые объекты с размером больше заданного. Например, вы можете выполнить этот запрос для идентификации программ, закрепляемых в базе данных средствами `DBMS_SHARED_POOL` (за дополнительной информацией об этом пакете обращайтесь к главе 24) для того, чтобы свести к минимуму подгрузку кода в SGA:

```

/* Файл в Сети: pssize.sql */
SELECT name, type, source_size, parsed_size, code_size
FROM user_object_size
WHERE code_size > &&1 * 1024
ORDER BY code_size DESC

```

Получение свойств хранимого кода

Представление `USER_PLSQL_OBJECT_SETTINGS` (появившееся в Oracle10g) содержит информацию о следующих параметрах компиляции хранимого объекта PL/SQL:

- `PLSQL_OPTIMIZE_LEVEL` — уровень оптимизации, использовавшийся при компиляции объекта.
- `PLSQL_CODE_TYPE` — режим компиляции объекта.
- `PLSQL_DEBUG` — признак отладочной компиляции.

- PLSQL_WARNINGS — настройки предупреждений, использовавшиеся при компиляции объекта.
- NLS_LENGTH_SEMANTICS — семантика длины NLS, использовавшаяся при компиляции объекта.

Пара примеров возможного применения этого представления:

- Поиск программ, не использующих все возможности оптимизирующего компилятора (уровень оптимизации 1 или 0):

```
/* Файл в Сети: low_optimization_level.sql */
SELECT owner, name
  FROM user_plsql_object_settings
 WHERE plsql_optimize_level IN (1,0);
```

- Проверка отключения предупреждений у хранимых программ:

```
/* Файл в Сети: disable_warnings.sql */
SELECT NAME, plsql_warnings
  FROM user_plsql_object_settings
 WHERE plsql_warnings LIKE '%DISABLE%';
```

В представлении USER_PROCEDURES перечисляются все функции и процедуры с их свойствами. В частности, в представление USER_PROCEDURES включается настройка модели AUTHID для программы (DEFINER или CURRENT_USER). Эта информация позволяет быстро определить, какие программы в пакете используют модель привилегий вызывающей стороны или создателя. Пример такого запроса:

```
/* Файл в Сети: show_authid.sql */
SELECT  AUTHID
        , p.object_name program_name
        , procedure_name subprogram_name
  FROM user_procedures p, user_objects o
 WHERE p.object_name = o.object_name
        AND p.object_name LIKE '<критерии имени программы или пакета>'
 ORDER BY AUTHID, procedure_name;
```

Анализ и изменение состояний триггеров

Запросы к триггерным представлениям (USER_TRIGGERS, USER_TRIG_COLUMNS) обычно используются для решения следующих задач:

- Включение или отключение всех триггеров для заданной таблицы. Вместо того чтобы писать код вручную, вы выполняете соответствующие команды DDL из кода PL/SQL. Пример такой программы приведен в разделе «Сопровождение триггеров» главы 19.
- Поиск триггеров, выполняемых при изменении некоторых столбцов, но не имеющих предложения секции WHEN. Следующий запрос поможет найти триггеры, не имеющие секции WHEN, которые являются источниками потенциальных проблем:

```
/* Файл в Сети: nowhen_trigger.sql */
SELECT *
  FROM user_triggers tr
 WHERE when_clause IS NULL AND
        EXISTS (SELECT 'x'
                  FROM user_trigger_cols
                 WHERE trigger_owner = USER
                   AND trigger_name = tr.trigger_name);
```

Анализ аргументов

Представление USER_ARGUMENTS может оказаться исключительно полезным для программиста: оно содержит информацию о каждом аргументе каждой хранимой программы

в вашей схеме. Оно одновременно предоставляет разнообразную информацию об аргументах в разобранном виде и запутанную структуру, с которой очень трудно работать. Простой сценарий SQL*Plus для вывода содержимого USER_ARGUMENTS для всех программ в заданном пакете:

```
/* Файл в Сети: descstest.sql */
SELECT object_name, argument_name, overload
       , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE package_name = UPPER ('&&1');
```

Более совершенная программа на базе PL/SQL для вывода содержимого USER_ARGUMENTS находится в файле show_all_arguments.sp на сайте книги.

Вы также можете создавать более конкретные запросы к представлению USER_ARGUMENTS для выявления возможных проблем с качеством кодовой базой. Например, Oracle рекомендует воздерживаться от использования типа LONG и использовать вместо него LOB. Кроме того, тип данных CHAR с фиксированной длиной может создать проблемы; намного лучше использовать VARCHAR2. Следующий запрос выявляет использование этих типов в определениях аргументов:

```
/* Файл в Сети: long_or_char.sql */
SELECT object_name, argument_name, overload
       , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE data_type IN ('LONG','CHAR');
```

Представление USER_ARGUMENTS может использоваться даже для получения информации о программах пакета, которую трудно получить другим способом. Предположим, я хочу получить список всех процедур и функций, определенных в спецификации пакета. Что вы говорите? «Нет проблем — просто выдать запрос к USER_PROCEDURES». И это был бы хороший ответ... вот только USER_PROCEDURES не сообщит вам, является ли программа функцией или процедурой (причем в зависимости от перегрузки она может быть и той и другой!)

Представление USER_ARGUMENTS содержит нужную информацию, но она хранится в далеко не очевидном формате. Чтобы определить, является ли программа функцией или процедурой, можно поискать в USER_ARGUMENTS строку данной комбинации «пакет/программа», у которой значение POSITION равно 0. Это значение Oracle использует для хранения «аргумента» RETURN функции. Если оно отсутствует, значит, программа должна быть процедурой.

Следующая функция использует эту логику для возвращения строки, обозначающей тип программы (если она перегружена для обоих типов, функция возвращает строку «FUNCTION, PROCEDURE»). Обратите внимание: функция list_to_string, используемая в теле функции, определяется в файле:

```
/* Файл в Сети: program_type.sf */
FUNCTION program_type (owner_in      IN VARCHAR2,
                      package_in     IN VARCHAR2,
                      program_in      IN VARCHAR2)
  RETURN VARCHAR2
IS
  c_function_pos  CONSTANT PLS_INTEGER := 0;

  TYPE type_aat IS TABLE OF all_objects.object_type%TYPE
    INDEX BY PLS_INTEGER;

  l_types          type_aat;
  retval           VARCHAR2 (32767);
```



```

BEGIN
    SELECT CASE MIN (position)
            WHEN c_function_pos THEN 'FUNCTION'
            ELSE 'PROCEDURE'
        END
    BULK COLLECT INTO l_types
    FROM all_arguments
    WHERE      owner = owner_in
            AND package_name = package_in
            AND object_name = program_in
    GROUP BY overload;

    IF l_types.COUNT > 0
    THEN
        retval := list_to_string (l_types, ',', distinct_in => TRUE);
    END IF;

    RETURN retval;
END program_type;

```

Наконец, следует сказать, что встроенный пакет DBMS_DESCRIBE предоставляет программный интерфейс PL/SQL, который возвращает почти ту же информацию, что и USER_ARGUMENTS. Впрочем, эти два механизма отличаются некоторыми особенностями работы с типами данных.

Анализ использования идентификаторов (Oracle Database 11g)

Проходит совсем немного времени, и рост объема и сложности кодовой базы создает серьезные проблемы с сопровождением и эволюцией. Допустим, мне потребовалось реализовать новую возможность в части существующей программы. Как убедиться в том, что я правильно оцениваю последствия от появления новой функции, и внести все необходимые изменения? До выхода Oracle Database 11g инструменты, которые могли использоваться для анализа последствий, в основном ограничивались запросами к ALL_DEPENDENCIES и ALL_SOURCE. Теперь, с появлением PL/Scope, я могу выполнять намного более подробный и полезный анализ.

PL/Scope собирает информацию об идентификаторах в исходном коде PL/SQL при компиляции кода и предоставляет собранную информацию в статических представлениях словарей данных. Собранная информация, доступная через USER_IDENTIFIERS, содержит очень подробные сведения о типах и использовании (включая объявления, ссылки, присваивание и т. д.) каждого идентификатора, а также о местонахождении использования в исходном коде.

Описание представления USER_IDENTIFIERS:

| Name | Null? | Type |
|------------------|----------|---------------|
| ----- | ----- | ----- |
| NAME | | VARCHAR2(128) |
| SIGNATURE | | VARCHAR2(32) |
| TYPE | | VARCHAR2(18) |
| OBJECT_NAME | NOT NULL | VARCHAR2(128) |
| OBJECT_TYPE | | VARCHAR2(13) |
| USAGE | | VARCHAR2(11) |
| USAGE_ID | | NUMBER |
| LINE | | NUMBER |
| COL | | NUMBER |
| USAGE_CONTEXT_ID | | NUMBER |

Вы можете писать запросы к USER_IDENTIFIERS для поиска в коде разнообразной информации, включая нарушения правил об именах. Такие редакторы PL/SQL, как Toad, наверняка скоро начнут предоставлять доступ к данным PL/Scope, упрощая их

использование для анализа кода. А пока этого не произошло, вам придется строить собственные запросы (или использовать написанные и опубликованные другими разработчиками).

Чтобы использовать PL/Scope, необходимо сначала приказать компилятору PL/SQL проанализировать идентификаторы программы в процессе компиляции. Для этого следует изменить значение параметра компилятора `PLSCOPE_SETTINGS`. Это можно делать на уровне сеанса и даже для отдельной программы, как в следующем примере:

```
ALTER SESSION SET plscope_settings='IDENTIFIERS:ALL'
```

Чтобы узнать значение `PLSCOPE_SETTINGS` для любой конкретной программы, обратитесь с запросом к `USER_PLSQL_OBJECT_SETTINGS`.

После включения PL/Scope при компиляции программы Oracle будет заполнять словарь данных подробной информацией об использовании каждого идентификатора в программе (переменные, типы, программы и т. д.).

Рассмотрим несколько примеров использования PL/Scope. Допустим, я создаю следующую спецификацию пакета и процедуру с включенной поддержкой PL/Scope:

```
/* Файл в Сети: 11g_plscope.sql */
ALTER SESSION SET plscope_settings='IDENTIFIERS:ALL'
/
CREATE OR REPLACE PACKAGE plscope_pkg
IS
    FUNCTION plscope_func (plscope_fp1 NUMBER)
        RETURN NUMBER;
    PROCEDURE plscope_proc (plscope_pp1 VARCHAR2);
END plscope_pkg;
/

CREATE OR REPLACE PROCEDURE plscope_proc1
IS
    plscope_var1    NUMBER := 0;
BEGIN
    plscope_pkg.plscope_proc (TO_CHAR (plscope_var1));
    DBMS_OUTPUT.put_line (SYSDATE);
    plscope_var1 := 1;
END plscope_proc1;
/
```

Настройки PL/Scope проверяются следующим образом:

```
SELECT name, plscope_settings
FROM user_plsql_object_settings
WHERE name LIKE 'PLSCOPE%'
```

| NAME | PLSCOPE_SETTINGS |
|---------------|------------------|
| PLSCOPE_PKG | IDENTIFIERS:ALL |
| PLSCOPE_PROC1 | IDENTIFIERS:ALL |

Проверка объявлений, обнаруженных в процессе компиляции этих двух программ:

```
SELECT name, TYPE
FROM user_identifiers
WHERE name LIKE 'PLSCOPE%' AND usage = 'DECLARATION' ORDER BY type, usage_id
```

| NAME | TYPE |
|--------------|-----------|
| PLSCOPE_FP1 | FORMAL IN |
| PLSCOPE_PP1 | FORMAL IN |
| PLSCOPE_FUNC | FUNCTION |
| PLSCOPE_PKG | PACKAGE |

```
PLSCOPE_PROC1    PROCEDURE
PLSCOPE_PROC     PROCEDURE
PLSCOPE_VAR1     VARIABLE
```

Теперь я могу получить информацию обо всех локально объявляемых переменных:

```
SELECT a.name variable_name, b.name context_name, a.signature
FROM user_identifiers a, user_identifiers b
WHERE   a.usage_context_id = b.usage_id
        AND a.TYPE = 'VARIABLE'
        AND a.usage = 'DECLARATION'
        AND a.object_name = 'PLSCOPE_PROC1'
        AND a.object_name = b.object_name ORDER BY a.object_type, a.usage_id
```

```
VARIABLE_NAME  CONTEXT_NAME  SIGNATURE
-----
PLSCOPE_VAR1   PLSCOPE_PROC1  401F008A81C7DCF48AD7B2552BF4E684
```

Впечатляет, однако возможности PL/Scope этим не ограничиваются. Я могу получить информацию обо всех местах программы, в которых используется эта переменная, а также о типе использования:

```
SELECT usage, usage_id, object_name, object_type
FROM user_identifiers sig
, (SELECT a.signature
    FROM user_identifiers a
    WHERE   a.TYPE = 'VARIABLE'
           AND a.usage = 'DECLARATION'
           AND a.object_name = 'PLSCOPE_PROC1') variables
WHERE sig.signature = variables.signature
ORDER BY object_type, usage_id
```

```
USAGE          USAGE_ID  OBJECT_NAME          OBJECT_TYPE
-----
DECLARATION    3          PLSCOPE_PROC1        PROCEDURE
ASSIGNMENT     4          PLSCOPE_PROC1        PROCEDURE
REFERENCE      7          PLSCOPE_PROC1        PROCEDURE
ASSIGNMENT     9          PLSCOPE_PROC1        PROCEDURE
```

Даже из этих простых примеров видно, что PL/Scope предоставляет выдающиеся возможности для того, чтобы лучше разобраться в коде и проанализировать изменения. Лукас Джеллема из AMIS предоставил более интересные и сложные примеры использования PL/Scope для проверки имен. Соответствующие запросы содержатся в файле `11g_plscope_amis.sql` на сайте книги.

Кроме того, я создал вспомогательный пакет и демонстрационные сценарии, которые помогут вам начать работу с PL/Scope. Просмотрите файлы `plscope_helper*.*`, а также другие файлы `plscope*.*`.

Управление зависимостями и перекомпиляция

Еще одной важной фазой компиляции и выполнения PL/SQL-программы является проверка ее *зависимостей*. Зависимость в PL/SQL представляет собой вид связи между программой и некоторым объектом Oracle, существующим вне этой программы. Серверные программы PL/SQL могут зависеть от таблиц, представлений, типов данных, процедур, функций, последовательностей и спецификаций пакетов, но не от тела пакетов или типов данных (последние относятся к «скрытой» реализации).

Основная цель проверки зависимостей в PL/SQL — не допустить выполнения программы, если хоть один из объектов, от которых она зависит, изменился с момента ее последней компиляции.

К счастью, управление зависимостями производится автоматически, от отслеживания зависимостей до их перекомпиляции в случае необходимости. Тем не менее некоторая ответственность за синхронизацию кода лежит на программистах, и в следующих разделах рассказывается, как, когда и для чего им следует воздействовать на этот процесс.

В Oracle10g и более ранних версиях зависимости отслеживались на уровне программных модулей. Если процедура зависела от функции пакета или столбца таблицы, то зависимой единицей становился пакет или таблица. Такой уровень детализации считался стандартным с первых дней PL/SQL и до недавнего времени.

В Oracle11g детализация отслеживания зависимостей была улучшена. Зависимости теперь отслеживаются не до уровня пакетов или таблиц, а до отдельных элементов (например, столбцов таблицы или программ пакета вместе с формальными параметрами вызова и режимами передачи). *Точное отслеживание зависимостей* означает, что программа останется действительной в случае добавления или перегрузки существующей программы в существующем пакете. Аналогичным образом при добавлении нового столбца в таблицу база данных не объявит недействительными все программы PL/SQL, ссылающиеся на эту таблицу, — только те программы, которые ссылаются на все столбцы (например, с использованием конструкции `SELECT *` или объявления `%ROWTYPE`). В следующих разделах эта ситуация рассматривается более подробно.

В разделе «Уточнение ссылок на переменные и столбцы в командах SQL» главы 3 приведен пример точного управления зависимостями.



К сожалению, в Oracle11g Release 2 эти данные все еще недоступны в представлениях словарей данных. Хочется верить, что в будущем информация станет доступной.

А пока использование любых версий, предшествующих Oracle11g, означает, что при любых изменениях в базах данных многие объекты будут автоматически становиться недействительными.

Анализ зависимостей с использованием представлений словаря данных

Для анализа зависимостей можно использовать некоторые представления словаря данных. Рассмотрим простой пример. Допустим, на сервере имеется пакет `bookworm`, а в нем имеется функция, извлекающая данные из таблицы `books`. Непосредственно после создания и таблица, и пакет действительны (`VALID`):

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE object_name = 'BOOKWORM';
```

| OBJECT_NAME | OBJECT_TYPE | STATUS |
|-------------|--------------|--------|
| BOOKWORM | PACKAGE | VALID |
| BOOKWORM | PACKAGE BODY | VALID |

При компиляции программы PL/SQL база данных формирует список объектов, необходимых для успешной компиляции пакета `BOOKWORM`. Для определения всех зависимостей между объектами можно построить граф зависимостей при помощи запроса к представлению `USER_DEPENDENCIES`:

```
SELECT name, type, referenced_name, referenced_type
FROM USER_DEPENDENCIES
WHERE name = 'BOOKWORM';
```

| NAME | TYPE | REFERENCED_NAME | REFERENCED_TYPE |
|----------|--------------|-----------------|-----------------|
| BOOKWORM | PACKAGE | STANDARD | PACKAGE |
| BOOKWORM | PACKAGE BODY | STANDARD | PACKAGE |
| BOOKWORM | PACKAGE BODY | BOOKS | TABLE |
| BOOKWORM | PACKAGE BODY | BOOKWORM | PACKAGE |

На рис. 20.1 эта информация представлена в виде ориентированного графа, на котором стрелки обозначают отношения типа «зависит от». Иначе говоря, на рис. 20.1 показано, что спецификация и тело пакета `bookworm` зависят от встроенного пакета `STANDARD` (см. врезку «Пакет `STANDARD`»), а тело пакета `bookworm` — от его же спецификации и от таблицы `books`.

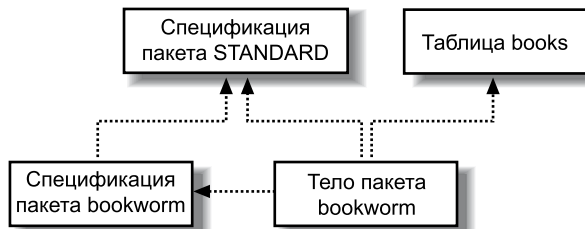


Рис. 20.1. Граф зависимостей пакета `bookworm`

Для отслеживания зависимостей Oracle записывает в словарь данных тело и спецификацию пакета как два разных элемента. Тело любого пакета зависит от его спецификации, но спецификация никогда не зависит от тела. От тела пакета не зависит ни один объект — собственно говоря, тела может и вовсе не быть.

Если вам приходилось достаточно долго заниматься сопровождением программного кода, вы знаете, что при подобном анализе учитываются не столько зависимости, сколько ссылки одних объектов на другие. Предположим, мне нужно изменить структуру таблицы `books`. Прежде всего необходимо выяснить, на какие из объектов это может повлиять:

```
SELECT name, type
FROM USER_DEPENDENCIES
WHERE referenced_name = 'BOOKS'
AND referenced_type = 'TABLE';
```

| NAME | TYPE |
|-----------|--------------|
| ADD_BOOK | PROCEDURE |
| TEST_BOOK | PACKAGE BODY |
| BOOK | PACKAGE BODY |
| BOOKWORM | PACKAGE BODY |
| FORMSTEST | PACKAGE |

Как видите, кроме пакета `bookworm` в схеме присутствуют и другие программы, ранее не упоминавшиеся. Но Oracle ничего не забывает!

И хотя Oracle очень тщательно отслеживает все зависимости, все предусмотреть невозможно: в словаре данных отслеживаются только зависимости локальных хранимых объектов, ссылки на которые оформлены в виде статических вызовов. Существует множество способов создания программ, которые не будут отражены в представлении `USER_DEPENDENCIES`. К их числу относятся внешние программы с интегрированным кодом

SQL или PL/SQL, удаленные хранимые процедуры, клиентские программы с вызовами локальных хранимых объектов и любые локальные программы, в которых используется динамический SQL.

Попробуем изменить структуру таблицы `books`, добавив в нее один новый столбец:

```
ALTER TABLE books MODIFY popularity_index NUMBER (8,2);
```

Oracle сразу же автоматически пометит все ссылки на таблицу `books` как недействительные (или в Oracle11g — только программные модули, в которых используется этот столбец). При любом изменении определения объекта, даже при его перекомпиляции без внесения изменений, все ссылки на объект также будут помечены как недействительные. Но на практике этот процесс еще сложнее. Если вы являетесь владельцем программы, выполняющей заданную команду DML со ссылкой на таблицу другой схемы, то после лишения вас привилегий на выполнение указанной команды вся программа помечается как недействительная.

Запрос к представлению `USER_OBJECTS` выводит следующую информацию:

```
/* Файл в Сети: invalid_objects.sql */
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

| OBJECT_NAME | OBJECT_TYPE | STATUS |
|-------------|--------------|---------|
| ADD_BOOK | PROCEDURE | INVALID |
| BOOK | PACKAGE BODY | INVALID |
| BOOKWORM | PACKAGE BODY | INVALID |
| FORMSTEST | PACKAGE | INVALID |
| FORMSTEST | PACKAGE BODY | INVALID |
| TEST_BOOK | PACKAGE BODY | INVALID |

Кстати, этот пример в очередной раз демонстрирует преимущества разбиения пакетов на две части: в большинстве случаев пометкой `INVALID` снабжаются тела пакетов, а не их спецификации. Если спецификация не изменяется, то и программные модули, зависящие от пакета, не будут объявлены недействительными. В данном примере недействительной объявлена только спецификация `FORMSTEST`, зависящая от таблицы `books`, потому что в ней используется объявление `books%ROWTYPE`.

И последнее замечание: для анализа программных зависимостей также можно воспользоваться процедурой Oracle `DEPTREE_FILL` в сочетании с представлениями `DEPTREE` или `IDeptree`.

Например, если запустить процедуру командой:

```
BEGIN DEPTREE_FILL('TABLE', USER, 'BOOKS'); END;
```

я смогу получить удобную информацию, запросив данные из представления `IDeptree`:

```
SQL> SELECT * FROM IDeptree;
```

```
DEPENDENCIES
```

```
-----
TABLE SCOTT.BOOKS
  PROCEDURE SCOTT.ADD_BOOK
  PACKAGE BODY SCOTT.BOOK
  PACKAGE BODY SCOTT.TEST_BOOK
  PACKAGE BODY SCOTT.BOOKWORM
  PACKAGE SCOTT.FORMSTEST
    PACKAGE BODY SCOTT.FORMSTEST
```

В листинге приведен результат рекурсивного запроса. Если вы захотите использовать эти объекты, выполните сценарий `$ORACLE_HOME/rdbms/admin/utldtree.sql` для построения

вспомогательной процедуры и представлений в вашей схеме. Или при желании смоделируйте его запросом вида:

```
SELECT RPAD (' ', 3*(LEVEL-1)) || name || ' (' || type || ') '
FROM user_dependencies
CONNECT BY PRIOR RTRIM(name || type) =
       RTRIM(referenced_name || referenced_type)
START WITH referenced_name = 'умя' AND referenced_type = 'mun'
```

Итак, теперь вы знаете, каким образом сервер хранит информацию об отношениях между объектами, и мы можем перейти к использованию этой информации в базе данных.

ПАКЕТ STANDARD

Почти в любой установке Oracle присутствует встроенный пакет STANDARD. Этот пакет создается вместе с представлениями словаря данных из catalog.sql и содержит многие базовые элементы языка PL/SQL, в том числе:

- функции (например, INSTR и LOWER);
- операторы сравнения (например, NOT, = (равно) и > (больше));
- заранее определенные исключения (например, DUP_VAL_ON_INDEX и VALUE_ERROR);
- подтипы (например, STRING и INTEGER).

Если вы захотите ознакомиться с исходным кодом пакета, откройте файл standard.sql, обычно находящийся в каталоге \$ORACLE_HOME/rdbms/admin.

Спецификация пакета STANDARD является корневым узлом графа зависимостей PL/SQL, то есть она не зависит от других программ PL/SQL (но большая часть программ PL/SQL зависит от нее). Пакет более подробно рассматривается в главе 24, «Архитектура PL/SQL».

Детализация зависимостей (Oracle11g)

Одной из самых замечательных возможностей PL/SQL является автоматизированный контроль зависимостей. Oracle автоматически отслеживает все объекты базы данных, от которых зависит программный модуль. Если какие-либо из этих объектов будут изменены, программный модуль помечается как недействительный, а его использование становится возможным только после перекомпиляции. Так, в примере с пакетом `scope_demo` при включении запроса из таблицы `employees` пакет помечается как зависимый от этой таблицы.

Как упоминалось ранее, в версиях, предшествующих Oracle11g, информация зависимостей сохранялась только на уровне объектов в целом. При внесении любых изменений в объект все зависимые от него программные модули объявляются недействительными, *даже если изменения на них никак не отражаются*.

Возьмем пакет `scope_demo`: он зависит от таблицы `employees`, но в нем используются только столбцы `department_id` и `salary`. В Oracle10g в случае изменения размера столбца `first_name` пакет помечался как недействительный.

В Oracle11g система отслеживания зависимостей была детализована до отдельных элементов внутри объектов. Для таблиц Oracle теперь отслеживает зависимости программного модуля от конкретных столбцов. Подобная детализация предотвращает лишние перекомпиляции и упрощает развитие кодовой базы приложения.

В Oracle Database 11g и выше я могу изменить размер столбца `first_name`, и этот столбец *не будет* помечен как недействительный:

```
ALTER TABLE employees MODIFY first_name VARCHAR2(2000)
/
Table altered.
SELECT object_name, object_type, status
  FROM all_objects
 WHERE owner = USER AND object_name = 'SCOPE_DEMO' /
```

| OBJECT_NAME | OBJECT_TYPE | STATUS |
|-------------|--------------|--------|
| SCOPE_DEMO | PACKAGE | VALID |
| SCOPE_DEMO | PACKAGE BODY | VALID |

Однако следует заметить, что без полного уточнения всех ссылок в переменных PL/SQL во встроенных командах SQL вы не сможете в полной мере пользоваться этим усовершенствованием.

А если говорить конкретно, уточнение имен переменных позволит избежать объявления программных модулей недействительными при добавлении новых столбцов в зависящую таблицу.

Возьмем исходную, неуточненную команду SELECT в `set_global`:

```
SELECT COUNT (*)
  INTO l_count
  FROM employees
 WHERE department_id = l_inner AND salary > l_salary;
```

В Oracle Database 11g вследствие детализации зависимостей база данных «заметит», что пакет `scope_demo` зависит только от `department_id` и `salary`.

Теперь предположим, что администратор базы данных добавит столбец в таблицу `employees`. Так как в команде SELECT присутствуют неуточненные ссылки на переменные PL/SQL, может оказаться, что имя нового столбца изменит информацию зависимостей для этого пакета. А именно, если имя нового столбца совпадает с неуточненной ссылкой в переменной PL/SQL, база данных разрешит ссылку на имя столбца. По этой причине база данных должна обновить информацию зависимостей для `scope_demo`, а это означает, что пакет должен быть объявлен недействительным.

И наоборот, если *уточнять* ссылки на все переменные PL/SQL во встроенных командах SQL, при компиляции программного модуля база данных будет знать об отсутствии возможных неоднозначностей. Даже при добавлении столбцов программа будет оставаться действительной (VALID).

Следует учесть, что список INTO запроса не является частью команды SQL. В результате переменные из списка не сохраняются в команде SQL, генерируемой компилятором PL/SQL. Соответственно, уточнение переменной именем области действия (или его отсутствие) не повлияет на анализ зависимостей базы данных.

Удаленные зависимости

Серверный код PL/SQL становится недействительным при любом изменении кода локального объекта, от которого он зависит. Однако если такой объект находится на другом компьютере, локальная копия Oracle не пытается объявить вызывающую программу PL/SQL недействительной в реальном времени. Вместо этого проверка откладывается до момента вызова.

Следующая программа зависит от процедуры `recompute_prices`, которая находится по ссылке `findat.idn.world`:

```
PROCEDURE synch_em_up (tax_site_in IN VARCHAR2, since_in IN DATE)
IS
BEGIN
```



```
IF tax_site_in = 'LONDON'
THEN
  recompute_prices@findat.ldn.world(cutoff_time => since_in);
END IF;
END;
```

Если удаленная процедура будет перекомпилирована, то при последующих попытках выполнить `synch_em_up`, скорее всего, будет получено сообщение об ошибке ORA-04062, уведомляющее об изменении временной метки (или сигнатуры) пакета `SCOTT.recompute_prices`. Если вызов остается действительным, Oracle перекомпилирует `synch_em_up`, и в случае успешного завершения следующий вызов будет выполнен успешно. Чтобы понять, как работает механизм вызова удаленных процедур, нужно знать, что компилятор PL/SQL хранит для каждой удаленной процедуры временную метку и сигнатуру.

- *Временная метка* — дата и время (с точностью до секунды) последнего изменения спецификации объекта, взятые из столбца `TIMESTAMP` представления `USER_OBJECTS`. В программах PL/SQL это не обязательно то же самое, что время последней компиляции, поскольку возможна перекомпиляция объекта без изменения его спецификации. (Указанный столбец имеет тип данных `DATE`, а не более новый тип `TIMESTAMP`.)
- *Сигнатура* — упрощенная спецификация объекта. Сигнатура определяет имя объекта, а также типы данных, порядок и режимы использования его параметров.

Таким образом, при компиляции процедуры `synch_em_up` Oracle извлекает временную метку и сигнатуру удаленной процедуры `recomputed_prices` и сохраняет их в байт-коде процедуры `synch_em_up`.

Дальше все просто: во время выполнения программы в зависимости от текущего значения параметра `REMOTE_DEPENDENCIES_MODE` Oracle сравнивает либо временную метку, либо сигнатуру удаленной процедуры со значением в байт-коде локальной процедуры, и если они не совпадают, выдает ошибку ORA-04062.

По умолчанию сравнивается временная метка, но иногда это ведет к ненужным перекомпиляциям. Администратор базы данных может изменить режим сравнения в системном инициализационном файле, а разработчик либо устанавливает его для текущего сеанса следующей командой:

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;
```

либо в коде PL/SQL:

```
EXECUTE IMMEDIATE 'ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE';
```

После этого до конца сеанса во всех программах PL/SQL будет использоваться метод проверки сигнатуры. Клиентские средства Oracle всегда выполняют команду `ALTER SESSION...SIGNATURE` сразу после подключения к базе данных, переопределяя настройки в инициализационном файле Oracle.

Oracle рекомендует использовать метод проверки сигнатуры в клиентских средствах (например, в Oracle Forms), а метод проверки временной метки — в межсерверных вызовах процедур. Однако учтите, что первый метод может стать причиной возникновения псевдоошибок в приложениях. Если исполняющее ядро ошибочно посчитает, что сигнатура программы не изменилась, Oracle не станет пометить как недействительную ту программу, из которой она вызывается. Тогда могут появиться трудноуловимые ошибки вычислений. Примеры опасных ситуаций такого рода:

- Изменение только значения по умолчанию одного из формальных параметров процедуры. Вызывающая программа будет продолжать использовать старое значение по умолчанию.
- Добавление в существующий пакет еще одного экземпляра перегруженной программы. Вызывающая сторона не активизирует новую версию, когда это потребуется.

- Изменение имени формального параметра. Если в вызывающей программе используется передача параметров по именам, могут возникнуть проблемы.

Во всех этих случаях приходится перекомпилировать вызывающую программу вручную. Метод же проверки временной метки, хотя иногда и приводит к лишней компиляции, гарантирует, что программа будет перекомпилирована в случае необходимости. Именно благодаря его надежности Oracle использует этот метод по умолчанию для межсерверных удаленных вызовов процедур.



Если вы пользуетесь методом проверки сигнатуры, Oracle рекомендует добавлять новые функции и процедуры в конец спецификации пакета, чтобы сократить количество ненужных перекомпиляций.

На практике сведение к минимуму перекомпиляций может существенно отразиться на доступности приложения. Дело в том, что базу данных можно «обмануть» и выдать локальный вызов за удаленный, чтобы использовать режим проверки сигнатуры. Следующий пример предполагает, что у вас имеется имя службы Oracle Net «localhost» с подключением к локальной базе данных:

```
CREATE DATABASE LINK loopback
  CONNECT TO bob IDENTIFIED BY swordfish USING 'localhost'
/
CREATE OR REPLACE PROCEDURE volatilecode AS
BEGIN
  -- ...
END;
/
CREATE OR REPLACE SYNONYM volatile_syn FOR volatilecode@loopback
/
CREATE OR REPLACE PROCEDURE save_from_recompile AS
BEGIN
  ...
  volatile_syn;
  ...
END;
/
```

Для использования этой конфигурации в рабочую систему включается вызов следующего вида:

```
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION SET REMOTE_DEPENDENCIES_MODE=SIGNATURE';
  save_from_recompile;
END;
/
```

При условии, что вы не делаете ничего, что бы изменяло сигнатуру `volatilecode`, код можно изменять и перекомпилировать без перехода `save_from_recompile` в недействительное состояние или ошибки времени выполнения. Вы даже можете перестроить синоним для совершенно другой процедуры. Такой подход не лишен недостатков; например, если `volatilecode` выводит что-либо средствами `DBMS_OUTPUT`, вы не увидите эту информацию, если только `save_from_recompile` не получает ее явно по каналу связи с базой данных с последующим непосредственным выводом. Но во многих приложениях такие обходные решения оказываются невысокой ценой за повышение доступности.

Ограничения модели удаленных вызовов Oracle

Вплоть до Oracle11g Release 2 программы PL/SQL не могли непосредственно использовать переменные и константы, курсоры и исключения, находящиеся на удаленных

серверах. Причем ограничение касается не только клиентского PL/SQL, вызывающего сервер базы данных, но и межсерверных вызовов удаленных процедур.

Простое обходное решение основано на использовании программ чтения/записи, инкапсулирующих нужные данные, тем более что это вообще считается проявлением хорошего стиля программирования. Однако следует заметить, что Oracle не позволяет обращаться к открытым переменным удаленных пакетов *даже косвенно*. Таким образом, если подпрограмма просто обращается к открытой переменной пакета, вы не сможете вызвать ее через связь с базой данных.

Обходное решение основано на инкапсуляции курсоров в подпрограммах открытия, выборки и закрытия. Например, если в спецификации серверного пакета `book_maint` объявлен курсор `book_cur`, в тело пакета можно поместить такой код:

```
PACKAGE BODY book_maint
AS
  prv_book_cur_status BOOLEAN;
  PROCEDURE open_book_cur IS
  BEGIN
    IF NOT book_maint.book_cur%ISOPEN
    THEN
      OPEN book_maint.book_cur;
    END IF;
  END;
  FUNCTION next_book_rec RETURN books%ROWTYPE
  IS
    l_book_rec books%ROWTYPE;
  BEGIN
    FETCH book_maint.book_cur INTO l_book_rec;
    prv_book_cur_status := book_maint.book_cur%FOUND;
    RETURN l_book_rec;
  END;

  FUNCTION book_cur_is_found RETURN BOOLEAN
  IS
  BEGIN
    RETURN prv_book_cur_status;
  END;
  PROCEDURE close_book_cur IS
  BEGIN
    IF book_maint.book_cur%ISOPEN
    THEN
      CLOSE book_maint.book_cur;
    END IF;
  END;
END book_maint;
```

К сожалению, этот подход не годится для перехвата и обработки удаленных исключений, поскольку исключения «тип данных» интерпретируются иначе, чем остальные типы данных. Вместо этого придется выполнить процедуру `RAISE_APPLICATION_ERROR` с пользовательскими номерами исключений от -20 000 до -20 999. О том, как написать пакет для перехвата и обработки исключений этого типа, рассказывалось в главе 6.

Перекомпиляция недействительных программ

Кроме недействительности в результате изменения задействованных объектов, новая программа может оказаться в недействительном состоянии в результате неудачной компиляции. В любом случае программа PL/SQL с пометкой `INVALID` не будет выполняться до тех пор, пока успешная перекомпиляция не изменит ее статус на `VALID`. Перекомпиляция может выполняться одним из трех способов:

1. **Автоматическая перекompиляция во время выполнения** — исполнительное ядро PL/SQL во многих обстоятельствах автоматически перекompилирует недействительную программу при ее вызове.
2. **Перекompиляция ALTER...COMPILE** — явная перекompиляция пакета командой ALTER.
3. **Перекompиляция уровня схемы** — использование многочисленных встроенных и внешних решений для перекompиляции всех недействительных программ в схеме или экземпляре базы данных.

Автоматическая перекompиляция во время выполнения

Так как Oracle хранит информацию о состоянии программных модулей, откомпилированных в базе данных, база данных известно, когда программный модуль становится недействительным и нуждается в перекompиляции. Когда пользователь, подключенный к базе данных, пытается выполнить (прямо или косвенно) недействительную программу, база данных автоматически пытается ее перекompилировать.

Тогда для чего может понадобиться явная перекompиляция? Есть две причины:

- В среде реальной эксплуатации перекompиляция по принципу «в нужный момент» может иметь негативные последствия — в отношении как ухудшения производительности, так и каскадного распространения недействительных объектов базы данных. Перекompиляция всех недействительных программ в то время, пока пользователи не работают с приложением, существенно улучшит их впечатления от работы с приложением (насколько это возможно).
- Перекompиляция программы, которая ранее была выполнена другим пользователем, подключенным к тому же экземпляру, может привести (и обычно приводит) к ошибкам следующего вида:

```
ORA-04068: existing state of packages has been discarded
ORA-04061: existing state of package "SCOTT.P1" has been invalidated
ORA-04065: not executed, altered or dropped package "SCOTT.P1"
ORA-06508: PL/SQL: could not find program unit being called
```

Такие ошибки происходят при перекompиляции пакета, обладающего *состоянием* (одна или несколько переменных, объявленных на уровне пакета). Все сеансы, ранее инициализировавшие этот пакет, теперь выходят из состояния синхронизации с откомпилированным пакетом. Когда база данных пытается обратиться к элементу такого пакета, она не может его найти и инициирует исключение.

Что делать? Ваше приложение *может* перехватить исключение, а затем снова вызвать ту же программу. Состояние пакета будет сброшено (на что указывает сообщение ORA-4068), а база данных сможет выполнить программу. К сожалению, в сеансе также будет сброшено состояние *всех пакетов*, включая DBMS_OUTPUT и другие встроенные пакеты. Крайне маловероятно, чтобы пользователи после этого смогли продолжить работу с приложением.

Для пользователей приложений на базе PL/SQL это означает то, что при каждой необходимости обновления (перекompиляции) кода все пользователи должны прервать работу с приложением. В современном мире «постоянно доступных» интернет-приложений такой сценарий *неприемлем*. В Oracle Database 11g Release 2 эта проблема наконец-то была решена за счет введения поддержки «оперативного обновления» кода приложения. Данная тема кратко рассматривается в конце главы.

Основной вывод по поводу автоматической перекompиляции заслуживает того, чтобы его повторить: до выхода Oracle Database 11g Release 2 в средах реальной эксплуатации

не следует делать *ничего*, что привело бы к перекомпиляции (автоматической или иной) любых хранимых объектов, к экземплярам которых сеанс может обратиться позднее. К счастью, средам разработки обычно не нужно беспокоиться о каскадных эффектах, а автоматическая перекомпиляция вне среды реальной эксплуатации способна существенно упростить задачу разработчика. И хотя перекомпиляция всех недействительных программ может быть полезна (см. далее), этот шаг не столь критичен.

Перекомпиляция ALTER...COMPILE

Программу, ранее откомпилированную в базе данных, всегда можно повторно откомпилировать командой ALTER...COMPILE. Например, в приведенном ранее примере по содержимому словаря данных можно было узнать, что три программных модуля стали недействительными.

Следующие команды перекомпилируют программы, чтобы их статус вернулся в состояние VALID:

```
ALTER PACKAGE bookworm COMPILE BODY REUSE SETTINGS;  
ALTER PACKAGE book COMPILE BODY REUSE SETTINGS;  
ALTER PROCEDURE add_book COMPILE REUSE SETTINGS;
```

Обратите внимание на секцию REUSE SETTINGS. Она гарантирует, что все настройки перекомпиляции (уровень оптимизации, уровень предупреждений и т. д.), ранее связанные с этой программой, останутся неизменными. При отсутствии REUSE SETTINGS после перекомпиляции будут применены текущие настройки сеанса.

Конечно, при большом количестве недействительных объектов вводить команды ALTER COMPILE одну за одной не хочется. Конечно, можно сгенерировать все команды ALTER простым запросом:

```
SELECT 'ALTER ' || object_type || ' ' || object_name  
      || ' COMPILE REUSE SETTINGS;  
FROM user_objects  
WHERE status = 'INVALID'
```

Но у такого решения есть один недостаток: перекомпиляция одного недействительного объекта может привести к тому, что несколько других объектов будут помечены как недействительные. Гораздо лучше воспользоваться другими, более сложными методами перекомпиляции всех недействительных программ; они будут рассмотрены ниже.

Перекомпиляция уровня схемы

Oracle предоставляет несколько способов перекомпиляции всех недействительных программ в конкретной схеме. Все перечисленные ниже инструменты должны запускаться с привилегиями SYSDBA, если явно не указано обратное. Все перечисленные файлы находятся в каталоге \$ORACLE_HOME/Rdbms/Admin.

- **utlip.sql** — объявляет недействительным и перекомпилирует весь код PL/SQL и представления во всей базе данных. (На самом деле создает некоторые структуры данных, объявляет объекты недействительными и предлагает перезапустить базу данных и выполнить **utlrip.sql**.)
- **utlrip.sql** — последовательно перекомпилирует все недействительные объекты. Хорошо подходит для однопроцессорных машин; на многопроцессорных машинах следует использовать **utlrcmp.sql**.
- **utlrcmp.sql** — как и **utlrip.sql**, перекомпилирует все недействительные объекты, но делает это параллельно; работа основана на отправке нескольких запросов на перекомпиляцию в очередь заданий базы данных. «Степень параллелизма» может передаваться

в целочисленном аргументе командной строки. Если аргумент не указан или равен 0, сценарий пытается выбрать подходящую степень параллелизма самостоятельно. Тем не менее Oracle предупреждает о том, что параллельная версия может не обеспечить кардинального выигрыша по производительности из-за конкуренции за возможность записи в системные таблицы.

- **DBMS_UTILITY.RECOMPILE_SCHEMA** — процедура существует начиная с Oracle8 Database и может запускаться из любой схемы; привилегии **SYSDBA** для этого не нужны. Процедура перекомпилирует программы в заданной схеме. Ее заголовок определяется следующим образом:

```
DBMS_UTILITY.COMPILE_SCHEMA (
    schema VARCHAR2
    , compile_all BOOLEAN DEFAULT TRUE,
    , reuse_settings BOOLEAN DEFAULT FALSE
);
```

- До выхода Oracle Database 10g процедура была реализована с ошибкой и часто объявляла недействительными столько новых программ, сколько перекомпилировала в состояние **VALID**. Похоже, теперь она работает как положено.
- **UTL_RECOMP** — этот встроенный пакет, появившийся в Oracle Database 10g, проектировался для обновлений или исправлений базы данных, требовавших значительной перекомпиляции. Он содержит две программы: одна перекомпилирует недействительные объекты последовательно, а другая использует **DBMS_JOB** для проведения параллельной перекомпиляции. Например, чтобы перекомпилировать все недействительные объекты в экземпляре базы данных в параллельном режиме, администратору достаточно ввести одну простую команду:

```
UTL_RECOMP.recomp_parallel
```

При выполнении параллельной версии пакет **DBMS_JOB** используется для организации очереди заданий на перекомпиляцию. Когда это происходит, все остальные задания в очереди временно приостанавливаются для предотвращения конфликтов перекомпиляции.

Пример вызова последовательной версии для перекомпиляции всех недействительных объектов в схеме **SCOTT**:

```
CALL UTL_RECOMP.recomp_serial ('SCOTT');
```

В системе с несколькими процессорами параллельная версия может ускорить перекомпиляцию. Но как указано в документации этого пакета, компиляция хранимых программ приводит к обновлению многих каталоговых структур и сопряжена с интенсивным вводом/выводом; вероятно, повышение скорости будет зависеть и от скорости ваших дисков.

Следующий пример запрашивает перекомпиляцию всех недействительных объектов в схеме **SCOTT**, с использованием до четырех параллельных потоков для перекомпиляции:

```
CALL UTL_RECOMP.recomp_parallel ('SCOTT', 4);
```



Соломон Якобсон, выдающийся администратор баз данных Oracle и специалист в области компьютерных технологий, также написал программу перекомпиляции, которая может использоваться для перекомпиляции всех недействительных программ в порядке зависимостей. Она работает с хранимыми программами, представлениями (включая материализованные), триггерами и пользовательскими объектными типами. Программа находится в файле `recompile.sql` на сайте книги.

ПРЕДОТВРАЩЕНИЕ ОБЪЯВЛЕНИЯ НЕДЕЙСТВИТЕЛЬНЫХ ЗАВИСИМОСТЕЙ

При изменении времени DDL объекта базы данных обычно база данных немедленно объявляет недействительными все его зависимости в локальной базе данных.

В Oracle Database 10g и последующих версиях, при перекомпиляции хранимой программы через сценарий ее исходного создания, зависимости не объявляются недействительными. Эта возможность не распространяется на перекомпиляции программ с использованием ALTER...COMPILE или автоматической перекомпиляции, с которой зависимости становятся недействительными. Обратите внимание: даже при использовании сценария база данных действует очень придирчиво; при любых изменениях в исходном коде — даже если изменить всего одну букву — зависимости программы помечаются как недействительные.

Предупреждения при компиляции

Предупреждения во время компиляции способны существенно упростить сопровождение вашего кода и снизить вероятность ошибок. Не путайте предупреждения компилятора с ошибками; с предупреждениями ваша программа все равно будет компилироваться и работать. Тем не менее при выполнении кода, для которого выдавались предупреждения, возможно неожиданное поведение или снижение производительности.

В этом разделе вы узнаете, как работают предупреждения компилятора и какие проблемы выявляются в текущих версиях. Начнем с краткого примера применения предупреждений времени компиляции в сеансе.

Пример

Очень полезное предупреждением компилятора PLW-06002 сообщает о наличии недостижимого кода. Рассмотрим следующую программу (см. файл `cantgothere.sql` на сайте книги). Так как переменная `salary` инициализируется значением 10 000, условная команда всегда будет отправлять меня на строку 9. Строка 7 выполняться не будет:

```
/* Файл в Сети: cantgothere.sql */
1  PROCEDURE cant_go_there
2  AS
3      l_salary NUMBER := 10000;
4  BEGIN
5      IF l_salary > 20000
6      THEN
7          DBMS_OUTPUT.put_line ('Executive');
8      ELSE
9          DBMS_OUTPUT.put_line ('Rest of Us');
10     END IF;
11     END cant_go_there;
```

Если откомпилировать этот код в любой версии до Oracle Database 10g, компилятор просто сообщит о том, что процедура создана. Но если включить предупреждения компиляции в сеансе этой или более поздней версии, то при попытке откомпилировать процедуру будет получен следующий ответ от компилятора:

SP2-0804: Procedure created with compilation warnings

```
SQL> SHOW err
Errors for PROCEDURE CANT_GO_THERE:
```

```
LINE/COL ERROR
-----
7/7      PLW-06002: Unreachable code
```

С этим предупреждением я могу вернуться к указанной строке, определить, почему она недостижима, и внести необходимые исправления.

Включение предупреждений компилятора

Oracle позволяет включать и отключать предупреждения компилятора, а также указывать, какие виды предупреждений представляют интерес. Предупреждения делятся на три категории:

- **Критичные** — ситуации, которые могут привести к неожиданному поведению или получению неверных результатов (как, например, проблемы с псевдонимами параметров).
- **Производительные** — ситуации, способные вызвать проблемы с производительностью (например, указание значения VARCHAR2 для столбца NUMBER в команде UPDATE).
- **Информационные** — ситуации, не влияющие на производительность или правильность выполнения кода, но которые стоит изменить ради того, чтобы упростить сопровождение.

Oracle позволяет включать и отключать предупреждения конкретной категории, всех категорий и даже конкретные предупреждения. Для этого используется команда ALTER DDL или встроенный пакет DBMS_WARNING.

Следующая команда включает предупреждения компиляции для системы в целом:

```
ALTER SYSTEM SET PLSQL_WARNINGS='string'
```

А следующая команда, например, включает предупреждения в вашей системе для всех категорий:

```
ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL';
```

Это значение особенно полезно во время разработки, потому что оно позволит обнаружить наибольшее количество потенциальных проблем в вашем коде.

Чтобы включить предупреждения в сеансе только для критичных проблем, введите следующую команду:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE';
```

А для изменения настройки предупреждений компилятора для конкретной, уже откомпилированной программы вводится команда следующего вида:

```
ALTER PROCEDURE hello COMPILE PLSQL_WARNINGS='ENABLE:ALL' REUSE SETTINGS;
```



Обязательно включите секцию REUSE SETTINGS, чтобы команда ALTER не влияла на все остальные настройки (например, уровень оптимизации).

Объединяя разные параметры, можно уточнять настройки с очень высоким уровнем детализации. Допустим, я хочу знать обо всех проблемах, относящихся к производительности, на данный момент не желаю отвлекаться на серверные проблемы, а предупреждение PLW-05005 (выход из функции без RETURN) должно рассматриваться как ошибка компиляции. Для этого вводится следующая команда:


```
ALTER SESSION SET PLSQL_WARNINGS=
  'DISABLE:SEVERE'
, 'ENABLE:PERFORMANCE'
, 'ERROR:05005';
```

Особенно полезна возможность интерпретации предупреждений как ошибок. Возьмем предупреждение PLW-05005; если оставить его без внимания при компиляции функции `no_return` (см. ниже), программа откомпилируется, и я смогу использовать ее в приложении:

```
SQL> CREATE OR REPLACE FUNCTION no_return
2   RETURN VARCHAR2
3   AS
4   BEGIN
5     DBMS_OUTPUT.PUT_LINE (
6       'Here I am, here I stay');
7   END no_return;
8   /
```

SP2-0806: Function created with compilation warnings

```
SQL> SHOW ERR
Errors for FUNCTION NO_RETURN:
```

LINE/COL ERROR

1/1 PLW-05005: function NO_RETURN returns without value at line 7

Если теперь изменить интерпретацию ошибки приведенной выше командой `ALTER SESSION` и перекомпилировать `no_return`, компилятор немедленно остановит попытку:

Warning: Procedure altered with compilation errors

Кстати говоря, настройки также можно изменить только для конкретной программы и пометить предупреждение как ошибку командой следующего вида:

```
ALTER PROCEDURE no_return COMPILE PLSQL_WARNINGS = 'error:6002' REUSE SETTINGS
/
```

Во всех этих разновидностях команды `ALTER` ключевое слово `ALL` может использоваться как простое и удобное обозначение всех категорий предупреждений:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

Oracle также предоставляет пакет `DBMS_WARNING` со сходными возможностями по установке и изменению параметров компиляции через PL/SQL API. В отличие от команды `ALTER`, `DBMS_WARNING` позволяет вносить изменения в конфигурацию тех предупреждений, которые вас интересуют, оставляя другие без изменений. Также после завершения работы можно легко восстановить исходные настройки.

Пакет `DBMS_WARNING` проектировался для использования в установочных сценариях, в которых может возникнуть необходимость отключения некоторых предупреждений или интерпретации предупреждения как ошибки для отдельных компилируемых программ. Может оказаться, что некоторые сценарии (внешние по отношению к тем, за которые вы отвечаете) вам неподконтрольны. Автор каждого сценария должен иметь возможность задать нужную конфигурацию предупреждений, наследуя более широкий спектр настроек из глобальной области действия.

Некоторые полезные предупреждения

В следующих разделах я представлю небольшую подборку предупреждений, реализованных Oracle, — с примерами кода, для которого они выдаются, и описаниями особенно интересного поведения.

Чтобы просмотреть полный список предупреждений для любой конкретной версии Oracle, найдите раздел *PLW* в книге *Error Messages* документации Oracle.

PLW-05000: несовпадение в NOCOPY между спецификацией и телом

Рекомендация NOCOPY сообщает базе данных Oracle, что вы, если это возможно, предпочли бы не создавать копии аргументов IN OUT. Отказ от копирования может повысить производительность программ, передающих большие структуры данных — например, коллекции или CLOB.

Рекомендация NOCOPY должна быть включена как в спецификацию, так и в тело программы (актуально для пакетов и объектных типов). Если рекомендация не присутствует в обоих местах, база данных применяет настройку, указанную в спецификации.

Пример кода, генерирующего это предупреждение:

```
/* Файл в Сети: plw5000.sql */
PACKAGE plw5000
IS
    TYPE collection_t IS
        TABLE OF VARCHAR2 (100);

    PROCEDURE proc (
        collection_in IN OUT NOCOPY
        collection_t);
END plw5000;

PACKAGE BODY plw5000
IS
    PROCEDURE proc (
        collection_in IN OUT
        collection_t)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE ('Hello!');
    END proc;
END plw5000;
```

Предупреждения компилятора отображаются в следующем виде:

```
SQL> SHOW ERRORS PACKAGE BODY plw5000
Errors for PACKAGE BODY PLW5000:
```

```
LINE/COL ERROR
-----
3/20      PLW-05000: mismatch in NOCOPY qualification between specification
          and body

3/20      PLW-07203: parameter 'COLLECTION_IN' may benefit from use of the
          NOCOPY compiler hint
```

PLW-05001: предыдущее использование строки противоречит этому использованию

Предупреждение проявляется при объявлении нескольких переменных или констант с одинаковыми именами. Оно также может проявиться в том случае, если список параметров программы, определенный в спецификации пакета, отличается от списка в определении из тела пакета.

Возможно, вы скажете: «Да, я видел эту ошибку, но это именно ошибка компиляции, а не предупреждение». Собственно, вы правы — следующая программа не откомпилируется:

```
/* Файл в Сети: plw5001.sql */
PROCEDURE plw5001
```

```

IS
  a  BOOLEAN;
  a  PLS_INTEGER;
BEGIN
  a := 1;
  DBMS_OUTPUT.put_line ('Will not compile');
END plw5001;

```

Компилятор выдает ошибку PLS-00371 (в разделе объявлений разрешено не более одного объявления 'A').

Почему же для этой ситуации создано *предупреждение*? Попробуем удалить присваивание переменной с именем a:

```

SQL> CREATE OR REPLACE PROCEDURE plw5001
2  IS
3      a  BOOLEAN;
4      a  PLS_INTEGER;
5  BEGIN
6      DBMS_OUTPUT.put_line ('Will not compile?');
7  END plw5001;
8  /

```

Procedure created.

Программа откомпилируется! База данных не выдает ошибку PLS-00371, потому что я не использую ни одну из переменных в своем коде. Предупреждение PLW-05001 устраняет этот недостаток, сообщая о том, что я объявляю одноименные переменные без использования:

```

SQL> ALTER PROCEDURE plw5001 COMPILE plsql_warnings = 'enable:all';
SP2-0805: Procedure altered with compilation warnings

```

```

SQL> SHOW ERRORS
Errors for PROCEDURE PLW5001:

```

LINE/COL ERROR

```

-----
4/4      PLW-05001: previous use of 'A' (at line 3) conflicts with this use

```

PLW-05003: фактический параметр с IN и NOCOPY может иметь побочные эффекты

Используя NOCOPY с параметром IN OUT, вы приказываете PL/SQL передавать аргумент по ссылке, а не по значению. Это означает, что любые изменения в аргументе вносятся непосредственно в переменную во внешней области действия. С другой стороны, при передаче «по значению» (ключевое слово NOCOPY отсутствует, или компилятор игнорирует рекомендацию NOCOPY) изменения вносятся в локальную копию параметра IN OUT. Когда программа завершается, изменения копируются в фактический параметр. (Если произойдет ошибка, измененные значения *не копируются* в фактический параметр.)

Рекомендация NOCOPY повышает вероятность *совмещения имен* аргументов, то есть ссылки на один блок памяти по двум разным именам.

Совмещение имен усложняет понимание и отладку кода; предупреждение компилятора, выявляющее эту ситуацию, будет чрезвычайно полезным.

Возьмем следующую программу:

```

/* Файл в Сети: plw5003.sql */
PROCEDURE very_confusing (
  arg1  IN          VARCHAR2
, arg2  IN OUT      VARCHAR2
, arg3  IN OUT NOCOPY VARCHAR2
)

```

```

IS
BEGIN
    arg2 := 'Second value';
    DBMS_OUTPUT.put_line ('arg2 assigned, arg1 = ' || arg1);
    arg3 := 'Third value';
    DBMS_OUTPUT.put_line ('arg3 assigned, arg1 = ' || arg1);
END;
```

Программа достаточно проста: передаются три строки, две из которых объявлены как IN OUT; аргументам IN OUT присваиваются значения; после каждого присваивания выводится значение первого аргумента IN. Теперь я запускаю процедуру и передаю одну локальную переменную во всех трех параметрах:

```

SQL> DECLARE
2   str   VARCHAR2 (100) := 'First value';
3   BEGIN
4       DBMS_OUTPUT.put_line ('str before = ' || str);
5       very_confusing (str, str, str);
6       DBMS_OUTPUT.put_line ('str after = ' || str);
7   END;
8   /
str before = First value
arg2 assigned, arg1 = First value
arg3 assigned, arg1 = Third value
str after = Second value
```

Хотя процедура `very_confusing` продолжает выполняться, присваивание `arg2` не отражается на значении аргумента `arg1`. Однако когда значение присваивается `arg3`, значение `arg1` (аргумент IN) заменяется на «Third value»! Более того, при завершении `very_confusing` присваивание `arg2` было применено к переменной `str`. Таким образом, при возврате управления во внешний блок переменной `str` присваивается значение «Second value», фактически заменяющее результат присваивания «Third value».

Как говорилось ранее, совмещение имен параметров может порождать очень запутанные ситуации. Если включить предупреждения компилятора, в таких программах, как `plw5003`, могут быть выявлены потенциальные проблемы совмещения имен:

```

SQL> CREATE OR REPLACE PROCEDURE plw5003
2   IS
3       str   VARCHAR2 (100) := 'First value';
4   BEGIN
5       DBMS_OUTPUT.put_line ('str before = ' || str);
6       very_confusing (str, str, str);
7       DBMS_OUTPUT.put_line ('str after = ' || str);
8   END plw5003;
9   /
```

SP2-0804: Procedure created with compilation warnings

```
SQL> SHOW ERR
```

Errors for PROCEDURE PLW5003:

LINE/COL ERROR

```

-----
6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects
6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects
```

PLW-05004: идентификатор также объявлен в пакете STANDARD или является встроенным в SQL

Многие разработчики PL/SQL не знают о пакете `STANDARD` и его влиянии на код PL/SQL. Например, многие программисты считают, что такие имена, как `INTEGER` и `TO_CHAR`,

являются зарезервированными словами языка PL/SQL. Однако на самом деле это тип данных и функция, объявленные в пакете STANDARD.

STANDARD — один из двух пакетов по умолчанию в PL/SQL (другой — DBMS_STANDARD). Поскольку STANDARD является пакетом по умолчанию, вам не нужно уточнять ссылки на такие типы данных, как INTEGER, NUMBER, PLS_INTEGER и т. д., именем STANDARD — хотя при желании это можно сделать.

Предупреждение PLW-5004 сообщает об объявлении идентификатора с таким же именем, как у элемента STANDARD (или встроенным именем SQL; многие встроенные имена, хотя и не все, объявляются в STANDARD).

Рассмотрим эту процедуру:

```

/* Файл в Сети: plw5004.sql
1  PROCEDURE plw5004
2  IS
3      INTEGER    NUMBER;
4
5      PROCEDURE TO_CHAR
6      IS
7      BEGIN
8          INTEGER := 10;
9      END TO_CHAR;
10 BEGIN
11     TO_CHAR;
12 END plw5004;
```

Для этой процедуры компилятор выводит следующие предупреждения:

LINE/COL ERROR

```

-----
3/4      PLW-05004: identifier INTEGER is also declared in STANDARD
         or is a SQL builtin
5/14     PLW-05004: identifier TO_CHAR is also declared in STANDARD
         or is a SQL builtin
```

Старайтесь избегать использования имен элементов, определенных в пакете STANDARD, если только у вас нет для этого очень веских причин.

PLW-05005: функция возвращает управление без значения

Очень полезное предупреждение — функция, не возвращающая значение, явно очень плохо спроектирована. Это одно из предупреждений, которые я бы рекомендовал интерпретировать как ошибку (синтаксис «ERROR:5005») в настройках PLSQL_WARNINGS.

Мы уже рассматривали один пример такой функции: no_return. Тот код был тривиальным; во всем исполняемом разделе не было ни одной команды RETURN. Конечно, код может быть и более сложным. Тот факт, что команда RETURN не выполняется, может быть скрыт за завесой сложной условной логики.

Впрочем, по крайней мере иногда в подобных ситуациях база данных способна обнаружить проблему, как в следующей программе:

```

1 FUNCTION no_return (
2     check_in IN BOOLEAN)
3     RETURN VARCHAR2
4 AS
5 BEGIN
6     IF check_in
7     THEN
8         RETURN 'abc';
9     ELSE
10        DBMS_OUTPUT.put_line (
```

```

11      'Here I am, here I stay');
12  END IF;
13  END no_return;

```

База данных обнаружила логическую ветвь, не приводящую к выполнению RETURN, поэтому для программы выдается предупреждение. Файл `plw5005.sql` на сайте книги содержит более сложную условную логику, которая демонстрирует, что предупреждение выдается и в более сложных программных структурах.

PLW-06002: недостижимый код

База данных Oracle теперь умеет проводить статический анализ программы для выявления строк кода, которые ни при каких условиях не получают управление во время выполнения. Это исключительно ценная информация, но иногда компилятор предупреждает о наличии проблемы в строках, которые на первый взгляд недостижимыми вовсе не являются. Более того, в описании действий, предпринимаемых для этой ошибки, говорится, что «предупреждение следует отключить, если большой объем кода был сделан недостижимым намеренно, а предупреждение приносит больше раздражения, чем пользы».

Пример такого предупреждения приводился ранее в разделе «Пример». Теперь рассмотрим следующий код:

```

/* Файл в Сети: plw6002.sql */
1  PROCEDURE plw6002
2  AS
3      l_checking BOOLEAN := FALSE;
4  BEGIN
5      IF l_checking
6      THEN
7          DBMS_OUTPUT.put_line ('Never here...');
8      ELSE
9          DBMS_OUTPUT.put_line ('Always here...');
10         GOTO end_of_function;
11     END IF;
12     <<end_of_function>>
13     NULL;
14 END plw6002;

```

В Oracle Database 10g и выше для этой программы выдаются следующие предупреждения:

```

LINE/COL ERROR
-----
5/7      PLW-06002: Unreachable code
7/7      PLW-06002: Unreachable code
13/4     PLW-06002: Unreachable code

```

Понятно, почему строка 7 помечена как недостижимая: `l_checking` присваивается значение FALSE, поэтому строка 7 выполняться не будет. Но почему строка 5 помечена как недостижимая? На первый взгляд этот код будет выполняться *всегда*! Более того, строка 13 тоже должна выполняться всегда, потому что GOTO передает управление этой строке по метке. И все же эта строка тоже помечена как недостижимая.

Такое поведение объясняется тем, что до выхода Oracle Database 11g предупреждение о недостижимости кода генерируется после его оптимизации. В Oracle Database 11g и выше анализ недостижимого кода стал намного более понятным и полезным.

Компилятор не вводит вас в заблуждение; говоря, что строка N недостижима, он сообщает, что она никогда не будет выполняться в соответствии со структурой оптимизированного кода.

Некоторые ситуации с недостижимым кодом не обнаруживаются компилятором. Пример:

```
/* Файл в Сети: plw6002.sql */
FUNCTION plw6002 RETURN VARCHAR2
AS
BEGIN
    RETURN NULL;
    DBMS_OUTPUT.put_line ('Never here...');
END plw6002;
```

Разумеется, вызов DBMS_OUTPUT.PUT_LINE недостижим, но в настоящее время компилятор не обнаруживает это обстоятельство — до версии 12.1.

PLW-07203: рекомендация NOCOPY может принести пользу в объявлении параметра

Как упоминалось ранее в отношении PLW-05005, использование NOCOPY для сложных, больших параметров IN OUT может улучшить производительность программ в некоторых условиях. Это предупреждение выдается для программ, у которых включение NOCOPY для параметров IN OUT может повысить эффективность выполнения. Пример такой программы:

```
/* Файл в Сети: plw7203.sql */
PACKAGE plw7203
IS
    TYPE collection_t IS TABLE OF VARCHAR2 (100);
    PROCEDURE proc (collection_in IN OUT collection_t);
END plw7203;
```

Это еще одно предупреждение, которое будет генерироваться во многих программах и вскоре начнет раздражать. Безусловно, предупреждение вполне справедливо, но в большинстве случаев последствия такой оптимизации останутся незамеченными. Более того, вряд ли вам удастся переключиться на NOCOPY без внесения изменений для обработки ситуаций с аварийным завершением программы, при котором данные остаются в неопределенном состоянии.

PLW-07204: преобразование типа столбца может привести к построению неоптимального плана запроса

Предупреждение выдается при вызове команд SQL из PL/SQL, при котором происходят неявные преобразования. Пример:

```
/* Файл в Сети: plw7204.sql */
FUNCTION plw7204
RETURN PLS_INTEGER
AS
    l_count PLS_INTEGER;
BEGIN
    SELECT COUNT(*) INTO l_count
    FROM employees
    WHERE salary = '10000';
    RETURN l_count;
END plw7204;
```

С этим предупреждением тесно связано предупреждение PLW-7202 (тип передаваемого параметра приводит к преобразованию типа столбца).

PLW-06009: обработчик OTHERS не завершается вызовом RAISE или RAISE_APPLICATION_ERROR

Это предупреждение (добавленное в Oracle Database 11g) выводится тогда, когда в обработчике исключений OTHERS не выполняется та или иная форма RAISE (повторное инициирование того же исключения или инициирование другого исключения), и не

вызывается `RAISE_APPLICATION_ERROR`. Другими словами, существует большая вероятность того, что программа «поглощает» исключение и игнорирует его. Ситуации, в которых ошибки действительно должны игнорироваться, встречаются довольно редко. Чаще исключение должно передаваться во внешний блок:

```
/* Файл в Сети: plw6009.sql */
FUNCTION plw6009
RETURN PLS_INTEGER
AS
    l_count    PLS_INTEGER;
BEGIN
    SELECT COUNT ( * ) INTO l_count
        FROM dual WHERE 1 = 2;

    RETURN l_count;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line ('Error!');
        RETURN 0;
END plw6009;
```

Тестирование программ PL/SQL

Всем нам нравится создавать что-то новое — собственно, это одна из причин, по которой мы занимаемся программированием. Мы берем интересную задачу и придумываем способ ее реализации на языке PL/SQL.

Однако никому не нравится возиться с тестированием своих программ (и писать документацию для них). Нам приходится это делать, но мы занимаемся этим без особого энтузиазма. На практике разработчики выполняют пару-тройку составленных на скорую руку тестов и решают, что если ошибки не найдены сразу, значит, их в программе нет. Почему это происходит?

- **Стремление к успеху.** Мы настолько убеждены в том, что наш код будет работать правильно, что предпочитаем держаться подальше от плохих новостей — и даже от самой возможности их появления. Мы проводим поверхностное тестирование, убеждаемся, что в первом приближении все работает, и ждем, пока другие найдут ошибки, если они есть (а они есть, можете не сомневаться!).
- **Сроки.** Сейчас время Интернета, время выхода на рынок определяет успех. Все должно быть готово немедленно — и мы выпускаем предварительную бета-версию как готовый продукт, а пользователи мучаются с тестированием наших разработок.
- **Некомпетентность руководства.** Как правило, руководители информационных отделов ничего не смыслят в разработке ПО. И если у вас нет времени на создание полноценного проекта (включая проектирование, написание кода, тестирование, документирование и т. д.), то в результате получится убогая поделка с множеством ошибок.
- **Затраты на организацию тестирования.** Создание тестовых программ обычно считается пустой тратой времени, ведь всегда найдется более важная работа. Одним из следствий такого подхода является то, что значительная часть обязанностей по тестированию перекладывается на отдел контроля качества (если он есть). Конечно, участие профессионалов в области контроля качества может оказать огромное влияние на результат, однако и разработчики не должны снимать с себя ответственность за модульное тестирование своего кода.

Таким образом, программы почти всегда нуждаются в дополнительном тестировании. Как повысить его эффективность в мире PL/SQL?

Для начала мы рассмотрим типичный пример неудачного процесса тестирования. Затем будут представлены выводы относительно ключевых проблем ручного тестирования, и мы познакомимся со средствами автоматизированного тестирования кода PL/SQL.

Типичные неэффективные технологии тестирования

В ходе тестирования программы необходимо определить, какие изменения вносятся в ходе ее работы: возвращаемая функцией строка, обновленная процедурой таблица и т. д. Затем вы заранее формируете прогноз правильного поведения программы для заданного набора входных данных и конфигурации (тестового сценария). После выполнения программы фактические результаты (внесенные программой изменения) сравниваются с прогнозируемыми значениями. Если они совпадают — значит, программа работает. Если что-то отличается — значит, где-то произошел сбой.

Это очень хорошее общее описание процесса тестирования; остается выяснить, как определить все необходимые тестовые сценарии и реализовать тесты. Начнем с весьма типичного (и к сожалению, неэффективного) подхода к тестированию.

Допустим, мы пишем большое приложение, обрабатывающее большое количество строк. В PL/SQL имеется функция SUBSTR, которая возвращает заданную часть строки. Однако ее использование связано с некоторыми проблемами. Дело в том, что эту функцию удобно применять, когда вы знаете начальную позицию и длину строки. Однако очень часто известно лишь местоположение начального (start) и конечного (end) символов, а длину строки приходится вычислять. Но по какой формуле? Чтобы не мучиться с вычислениями (кстати говоря, правильный ответ $\text{end} - \text{start} + 1$), мы напомним функцию betwnstr, которая произведет все вычисления за нас:

```
/* Файл в Сети: betwnstr.sf */
FUNCTION betwnstr (string_in IN VARCHAR2
                  , start_in IN INTEGER
                  , end_in IN INTEGER
)
RETURN VARCHAR2
IS
BEGIN
RETURN (SUBSTR ( string_in, start_in, end_in - start_in + 1));
END betwnstr;
```

К сожалению, объем работы весьма велик, а это всего лишь одна из множества написанных программ, которую необходимо протестировать. Мы пишем примитивный «тестовый сценарий» с использованием DBMS_OUTPUT.PUT_LINE и запускаем его:

```
BEGIN
  DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', 3, 5)
                              , '**Really NULL**'));
END;
```

cde

Работает... надо же! Но одного теста недостаточно. Давайте зададим конечное значение за пределами строки — например, 500. Функция должна вернуть остаток строки, как бы это сделала функция SUBSTR:

```
BEGIN
  DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', 3, 500)
                              , '**Really NULL**'));
END;
```

cdefg

Снова работает! Теперь нужно убедиться в том, что функция правильно работает со значениями NULL:

```
BEGIN
  DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', NULL, 5)
                                , '**Really NULL**'));
END;

**Really NULL** ) ;
```

Три из трех! Функция работает правильно, скажете вы? Нет — скорее всего, вы покачаете головой и скажете себе: «Такое тестирование даже в первом приближении не проверяет все возможные сценарии. Оно даже не изменяет значение первого аргумента! К тому же при каждом изменении входных значений предыдущий тест терялся».

И это будет правильно. Вместо того чтобы наугад проверять разные значения аргументов, следует составить список тестовых сценариев, поведение которых мы хотим проверить.

| String | Start | End | Результат |
|------------------|---------------------|-----------------------------|-----------|
| abcdefg | 1 | 3 | abc |
| abcdefg | 0 | 3 | abc |
| <любое значение> | NULL | Не NULL | NULL |
| <любое значение> | Не NULL | NULL | NULL |
| NULL | <любое значение> | <любое значение> | NULL |
| abcdefg | Положительное число | Число, меньшее start | NULL |
| abcdefg | 1 | Число, большее длины строки | abcdefg |

На основании этой таблицы строится простой сценарий следующего вида:

```
/* Файл в Сети: betwnstr.tst */
BEGIN
  DBMS_OUTPUT.put_line ('Test 1: ' || betwnstr (NULL, 3, 5));
  DBMS_OUTPUT.put_line ('Test 2: ' || betwnstr ('abcdefg', 0, 5));
  DBMS_OUTPUT.put_line ('Test 3: ' || betwnstr ('abcdefg', 3, 5));
  DBMS_OUTPUT.put_line ('Test 4: ' || betwnstr ('abcdefg', -3, -5));
  DBMS_OUTPUT.put_line ('Test 5: ' || betwnstr ('abcdefg', NULL, 5));
  DBMS_OUTPUT.put_line ('Test 6: ' || betwnstr ('abcdefg', 3, NULL));
  DBMS_OUTPUT.put_line ('Test 7: ' || betwnstr ('abcdefg', 3, 100));
END;
```

Каждый раз, когда нам потребуется протестировать `betwnstr`, мы просто выполним этот сценарий и проверим результаты. Для исходной реализации они будут выглядеть так:

```
SQL> @betwnstr.tst
Test 1:
Test 2: abcdef
Test 3: cde
Test 4:
Test 5:
Test 6:
Test 7: cdefgh
```

«Проверим результаты»... Легко сказать, но как это сделать? Как узнать, прошли ли тесты? Нам придется просматривать результаты строку за строкой и проверять их по таблице. К тому же при действительно тщательном тестировании у нас будет более 30 тестовых сценариев (не забыли об отрицательных значениях начальной и конечной позиции?). На просмотр результатов уйдет несколько минут, и это для совершенно тривиального кода. Сама мысль о распространении этой методики на «реальный» код выглядит устрашающе. А теперь представьте, что программа изменяет две таблицы и возвращает два аргумента OUT. Счет тестовых сценариев пойдет на сотни, да еще добавьте к этому проблемы настройки конфигурации и проверку правильности содержимого таблиц.

И все же многие разработчики выбирают этот путь при «тестировании» своего кода. Почти все проводимое тестирование обладает рядом ключевых недостатков:

- **Тестовый код пишется вручную**, что существенно ограничивает объем тестирования. У кого найдется время на написание всего необходимого кода?
- **Неполное тестирование**. Будем откровенны: при тестировании мы обычно ограничиваемся несколькими очевидными случаями, чтобы убедиться в том, что программа не имеет очевидных изъянов. Вряд ли это можно назвать полноценным тестированием.
- **Одноразовые тесты**. Было бы неплохо подумать о будущем и понять, что нам (или кому-то другому) еще неоднократно придется проводить те же тесты.
- **Ручная проверка**. Если мы будем полагаться только на собственную наблюдательность при проверке результатов, это займет слишком много времени и, скорее всего, приведет к ошибкам.
- **Тестирование после разработки**. Многие программисты говорят себе: «Вот допишу программу и займусь тестированием». Вроде бы вполне разумный подход, и все же он в корне неверен. Во-первых, наши программы никогда не «дописываются» — исправления вносятся до последней минуты, поэтому времени на тестирование вечно не хватает. Во-вторых, если вы начинаете думать о тестировании только после завершения работы над реализацией, вы подсознательно выбираете тесты, которые имеют наибольшую вероятность успеха, и избегаете тех, которые могут столкнуться с проблемами. Так уж устроено наше сознание.

Если вы хотите, чтобы тестирование было действительно эффективным и тщательным, необходимо действовать иначе. Тесты должны определяться так, чтобы они могли легко сопровождаться с течением времени. Мы должны иметь возможность легко проводить тестирование и, что еще важнее, — без продолжительного анализа определить результат: успех или неудача. При этом выполнение тестов не должно требовать написания больших объемов тестового кода.

Далее я сначала дам несколько советов относительно того, как организовать тестирование вашего кода. Затем будут рассмотрены средства автоматизации тестирования для разработчиков PL/SQL; особое внимание будет уделено `utPLSQL` и `Quest Code Tester for Oracle`.

Общие рекомендации по тестированию кода PL/SQL

Выбор инструмента тестирования зависит только от вас, но для повышения качества тестирования следует принять во внимание следующие факторы:

- **Осознанная необходимость тестирования**. Самые важные изменения должны прозойти у вас в голове. От позиции «Надеюсь, эта программа работает» необходимо перейти к позиции «Я должен быть способен доказать, что моя программа работает». Осознав необходимость тестирования, вы начнете писать более модульный код, который проще тестируется. Также вам понадобятся инструменты, позволяющие более эффективно проводить процесс тестирования.
- **Продумайте тестовые сценарии до того, как возьметесь за написание программы**, — сформулируйте их на листке бумаги или в программе, управляющей процессом тестирования. Очень важно, чтобы ваши представления о том, что же необходимо протестировать, нашли внешнее воплощение; в противном случае вы с большой вероятностью о чем-нибудь забудете. Приступая к работе над программой в понедельник, я легко могу представить 25 разных сценариев (требований), которые необходимо реализовать. Три дня спустя отведенное время заканчивается, я перехожу к тестированию — и как ни удивительно, могу вспомнить всего пять тестовых сценариев (да и то самых очевидных). Если вы составите список известных тестовых сценариев в начале работы, вам будет намного проще запомнить и проверить их.

- **Не беспокойтесь о стопроцентном тестовом покрытии.** Вряд ли существует хоть одна нетривиальная программа, которая была полностью протестирована. Не ставьте себе цель обеспечить стопроцентное покрытие всех возможных тестовых сценариев. Скорее всего, это нереально, а результат только приведет вас в уныние. Самое важное в тестировании — взяться за него. И что с того, что в фазе 1 было реализовано всего 10% тестовых сценариев? Это на 10% больше, чем было прежде. А когда ваши тестовые сценарии (и связанный с ними код) окажутся на месте, дополнить их новыми возможностями станет проще.
- **Интеграция тестирования в разработку.** Нельзя откладывать тестирование до того момента, когда вы «закончите» работу над программой. Чем раньше вы начнете думать над ним, тем лучше. Составьте перечень тестовых сценариев, спланируйте тестовый код и запускайте тесты в процессе реализации, отладки и совершенствования программы. При любой возможности снова выполняйте тесты и убеждайтесь в том, что программа действительно движется вперед. Если вам нужно красивое название (методология), которое убедит вас в ценности такого подхода, изучите широко распространенную (в объектно-ориентированных кругах) методологию *разработки через тестирование* (TDD, Test-Driven Development).
- **Подготовьте регрессионные тесты.** Все перечисленные предложения в сочетании с инструментами, о которых рассказано в следующем разделе, помогут вам организовать регрессионное тестирование. Такие тесты должны предотвратить регрессию, то есть нарушение работоспособности ранее работавшего кода при внесении изменений. Ужасно неприятно, когда при выпуске версии 2 продукта половина функций версии 1 вдруг перестает работать. «Как это могло произойти?» — вопрошают пользователи. И что им на это ответить? Честный ответ должен звучать так: «Извините, но у нас не было времени на регрессионные тесты. Когда мы вносим изменения в своем запутанном коде, на самом деле мы понятия не имеем, что при этом могло сломаться». Но после создания нормальных регрессионных тестов вы можете уверенно вносить изменения и выдавать новые версии.

Средства автоматизации тестирования программ PL/SQL

В наши дни разработчики PL/SQL могут использовать целый ряд автоматизированных инфраструктур и средств тестирования своего кода:

- **utPLSQL** — первая тестовая инфраструктура для PL/SQL, фактически «JUnit для PL/SQL». utPLSQL реализует принципы тестирования, принятые в методологии *экстремального программирования*, и автоматически выполняет написанный вручную тестовый код с проверкой результатов. В следующем разделе описан пример сеанса тестирования с использованием utPLSQL. За полной информацией обращайтесь по адресу <http://utplsql.sourceforge.net>.
- **Code Tester for Oracle** — коммерческая программа тестирования с самым высоким уровнем автоматизации тестов. Тестовый код генерируется на основании поведения, определяемого в пользовательском интерфейсе, после чего программа выполняет тесты и выводит результаты. Предоставляется Dell в составе пакета Toad Development Suite.
- **Oracle SQL Developer** — бесплатная среда разработки от Oracle со средствами интеграции модульного тестирования, близкий аналог Code Tester for Oracle.
- **PLUTO** — аналог utPLSQL, реализованный с использованием объектных типов Oracle.
- **dbFit** — эта инфраструктура использует совершенно иной подход к определению тестов: она строится на базе табличных сценариев и позволяет применять тесты FIT/FitNesse непосредственно к базе данных.

Трассировка кода PL/SQL

Ваша программа компилируется успешно, но в ходе выполнения тестового сценария происходит сбой: в программе где-то допущена ошибка. Как найти ее причину? Конечно, можно сразу взяться за отладчик (практически в любом редакторе PL/SQL имеется визуальный отладчик с возможностью установки точек прерывания и отслеживания состояния данных), но сначала стоит провести трассировку выполнения программы.

Прежде чем рассматривать возможности трассировки, стоит сказать пару слов о различиях между отладкой и трассировкой. Разработчики часто смешивают эти два понятия, однако это далеко не одно и то же. В двух словах: вы сначала выполняете трассировку для получения расширенной информации о поведении приложения, которая помогает выявить источник проблем, а затем используете отладчик для поиска конкретных строк кода, порождающих ошибку.

Ключевое различие между трассировкой и отладкой заключается в том, что трассировка выполняется в «пакетном» режиме, а отладка является интерактивным процессом. Я включаю режим трассировки и выполняю код своего приложения. Когда его выполнение завершится, я открываю журнал и использую полученную информацию для проведения сеанса отладки. Во время отладки я выполняю код строку за строкой (обычно начиная с точки прерывания, близкой к источнику проблемы). Сеансы отладки обычно получаются слишком долгими и утомительными, поэтому стоит сделать все возможное, чтобы свести к минимуму время, проведенное за отладкой. Основательная упреждающая трассировка поможет вам в этом.

Каждое приложение должно включать определенные программистом трассировочные вызовы. В этом разделе представлены основные средства трассировки PL/SQL, но сначала нужно упомянуть принципы, которые должны соблюдаться при реализации трассировки:

- Трассировочные вызовы должны оставаться в коде на протяжении всех фаз разработки и развертывания. Иначе говоря, трассировочные вызовы, вставленные в ходе разработки, не следует удалять при переходе приложения в фазу реальной эксплуатации. Трассировка часто помогает понять, что происходит с вашим приложением при выполнении в реальной среде.
- Сведите затраты на вызов трассировочных средств к абсолютному минимуму. При отключенной трассировке их присутствие в коде не должно никак отражаться на быстродействии приложения.
- Не вызывайте программу `DBMS_OUTPUT.PUT_LINE` для выполнения трассировки прямо в коде приложения. Эта встроенная программа не обладает достаточной гибкостью или мощностью для качественной трассировки.
- Вызовите `DBMS_UTILITY.FORMAT_CALL_STACK` или подпрограмму `UTL_CALL_STACK` (12с и выше) для сохранения стека вызовов в трассировочных данных.
- Обеспечьте возможность простого включения и отключения трассировки в вашем коде. Включение трассировки не должно требовать вмешательства службы поддержки. Также не следует поставлять отдельную версию приложения с включенной трассировкой.
- Если кто-то уже создал механизм трассировки, который вы можете использовать в своей работе (и который соответствует этим принципам), не тратьте время на создание собственного механизма трассировки.

Начнем с последнего принципа. Какие средства трассировки PL/SQL существуют в настоящее время?

- **DBMS_APPLICATION_INFO** — этот встроенный пакет предоставляет API, при помощи которого приложения «регистрируют» свое текущее состояние выполнения

в базе данных Oracle. Трассировочная информация записывается в динамические представления `v$`. Подробное описание приводится в следующем разделе.

- **Log4PLSQL** — инфраструктура с открытым исходным кодом, построенная по образцу (и на основе) `log4J`, очень популярного механизма протоколирования для языка Java. За дополнительной информацией обращайтесь по адресу <http://log4plsql.sourceforge.net>.
- **opp_trace** — этот пакет, доступный на сайте книги, предоставляет простую, но эффективную функциональность трассировки.
- **DBMS_TRACE** — это встроенное средство обеспечивает трассировку кода PL/SQL, но не позволяет включать в результаты какие-либо данные приложения. С другой стороны, `DBMS_TRACE` может использоваться без модификации исходного кода программы.



Вы также можете воспользоваться одним из встроенных профайлеров PL/SQL для получения информации о профилях производительности каждой строки и подпрограммы вашего приложения. Профайлеры рассматриваются в главе 21.

DBMS_UTILITY.FORMAT_CALL_STACK

Функция `DBMS_UTILITY.FORMAT_CALL_STACK` возвращает отформатированную строку с содержимым стека вызовов: последовательностью вызовов процедур или функций, приведшей к точке вызова функции. Иначе говоря, эта функция отвечает на вопрос: «Как я сюда попал?»

Следующий пример демонстрирует использование этой функции и показывает, как выглядит отформатированная строка:

```
/* Файл в Сети: callstack.sql */
SQL> CREATE OR REPLACE PROCEDURE proc1
  2  IS
  3  BEGIN
  4      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_call_stack);
  5  END;
  6  /
```

Procedure created.

```
SQL> CREATE OR REPLACE PACKAGE pkg1
  2  IS
  3      PROCEDURE proc2;
  4  END pkg1;
  5  /
```

Package created.

```
SQL> CREATE OR REPLACE PACKAGE BODY pkg1
  2  IS
  3      PROCEDURE proc2
  4      IS
  5      BEGIN
  6          proc1;
  7      END;
  8  END pkg1;
  9  /
```

Package body created.

```

SQL> CREATE OR REPLACE PROCEDURE proc3
2  IS
3  BEGIN
4      FOR indx IN 1 .. 1000
5      LOOP
6          NULL;
7      END LOOP;
8
9      pkg1.proc2;
10 END;
11 /

```

Procedure created.

```

SQL> BEGIN
2  proc3;
3  END;
4  /
----- PL/SQL Call Stack -----
      object      line object
      handle      number name
000007FF7EA83240      4 procedure HR.PROC1
000007FF7E9CC3B0      6 package body HR.PKG1
000007FF7EA0A3B0      9 procedure HR.PROC3
000007FF7EA07C00      2 anonymous block

```

О чем необходимо помнить при использовании DBMS_UTILITY.FORMAT_CALL_STACK:

- При вызове подпрограммы из пакета в отформатированном стеке вызовов выводится только имя пакета, но не имя подпрограммы (после того, как код был откомпилирован, все имена «теряются», то есть становятся недоступными для исполнительного ядра).
- Чтобы просмотреть фактический исполняемый код, создайте запрос к ALL_SOURCE с именем программного модуля и номером строки.
- Когда вы отслеживаете процесс выполнения (см. подробности далее) или регистрируете ошибки, вызовите эту функцию и сохраните вывод в журнале для последующего просмотра и анализа.
- Разработчики уже давно просили Oracle предоставить средства для разбора строки, и в Oracle Database 12c наконец-то появился новый пакет UTL_CALL_STACK. (Если вы еще не перешли на версию 12.1 и выше, обратитесь к файлу callstack.pkg — он предоставляет аналогичную функциональность.)

UTL_CALL_STACK (Oracle Database 12c)

Oracle Database 12c предоставляет пакет UTL_CALL_STACK для получения информации о выполняемых подпрограммах. Хотя имя пакета создает впечатление, что пакет только предоставляет информацию о стеке вызовов, также предоставляется доступ к стеку ошибок и стеку трассировки.

Каждый стек обладает глубиной (индикатором позиции в стеке); вы можете запросить информацию на определенной глубине всех трех видов стеков, с которыми работает пакет. Одним из главных усовершенствований UTL_CALL_STACK по сравнению с DBMS_UTILITY.FORMAT_CALL_STACK стала возможность получения уточненных имен с именами программных модулей, всех лексических родителей подпрограммы и имени самой подпрограммы. Впрочем, для стека трассировки эта дополнительная информация недоступна.

Ниже приведена таблица подпрограмм пакета с парой примеров.

| Имя | Описание |
|------------------------|--|
| BACKTRACE_DEPTH | Возвращает количество элементов в стеке трассировки |
| BACKTRACE_LINE | Возвращает номер строки для заданной глубины стека трассировки |
| BACKTRACE_UNIT | Возвращает имя программного модуля для заданной глубины стека трассировки |
| CONCATENATE_SUBPROGRAM | Возвращает полное имя, полученное посредством конкатенации |
| DYNAMIC_DEPTH | Возвращает количество подпрограмм в стеке вызовов, включая SQL, Java и другие не относящиеся к PL/SQL контексты. Например, если А вызывает В, затем В вызывает С и С вызывает В, то стек будет выглядеть следующим образом (внизу выводится динамическая глубина): А В С В 4 3 2 1 |
| ERROR_DEPTH | Возвращает количество ошибок в стеке вызовов |
| ERROR_MSG | Возвращает сообщение об ошибке для заданной глубины стека ошибок |
| ERROR_NUMBER | Возвращает код ошибки для заданной глубины стека ошибок |
| LEXICAL_DEPTH | Возвращает лексическую глубину вложенности подпрограммы для заданной динамической глубины |
| OWNER | Возвращает имя владельца программного модуля подпрограммы на заданной динамической глубине |
| SUBPROGRAM | Возвращает полное имя подпрограммы для заданной динамической глубины |
| UNIT_LINE | Возвращает номер строки модуля для подпрограммы с заданной динамической глубиной |

Несколько типичных применений UTL_CALL_STACK:

- Получение полного имени подпрограммы.** Вызов `UTL_CALL_STACK.SUBPROGRAM` возвращает массив строк; функция `CONCATENATE_SUBPROGRAM` получает этот массив и возвращает одну строку с именами, разделенными точками. В следующем фрагменте при вызове `SUBPROGRAM` передается 1, потому что я хочу получить информацию о программе на вершине стека — текущей выполняемой подпрограмме:

```
/* Файл в Сети: 12c_utl_call_stack.sql */
```

```
CREATE OR REPLACE PACKAGE pkg1
```

```
IS
```

```
    PROCEDURE proc1;
```

```
END pkg1;
```

```
/
```

```
CREATE OR REPLACE PACKAGE BODY pkg1
```

```
IS
```

```
    PROCEDURE proc1
```

```
    IS
```

```
        PROCEDURE nested_in_proc1
```

```
        IS
```

```
        BEGIN
```

```
            DBMS_OUTPUT.put_line (
                utl_call_stack.concatenate_subprogram (utl_call_stack.subprogram (1)));
```

```
        END;
```

```
    BEGIN
```

```
        nested_in_proc1;
```

```
    END;
```

```
END pkg1;
```

```
/
```

```
BEGIN
```

```
    pkg1.proc1;
```

```
END;
```

```
/
```

```
PKG1.PROC1.NESTED_IN_PROC1
```


2. Вывод имени программного модуля и номера строки позиции, в которой было инициировано текущее исключение. Сначала создается функция `BACKTRACE_TO`, которая «скрывает» вызовы подпрограмм `UTL_CALL_STACK`. Во всех вызовах `BACKTRACE_UNIT` и `BACKTRACE_LINE` передается значение, возвращаемое функцией `ERROR_DEPTH`. Значение глубины для ошибок отличается от значения для стека вызовов: в стеке вызовов 1 соответствует вершине стека (текущей выполняемой подпрограмме). В стеке ошибок позиция кода, в которой была инициирована ошибка, находится на глубине `ERROR_DEPTH`, а не на глубине 1:

```
/* Файл в Сети: 12c_utl_call_stack.sql */
```

```
CREATE OR REPLACE FUNCTION backtrace_to
RETURN VARCHAR2
IS
BEGIN
    RETURN    utl_call_stack.backtrace_unit (utl_call_stack.error_depth)
            || ' line '
            || utl_call_stack.backtrace_line (utl_call_stack.error_depth);
END;
```

```
CREATE OR REPLACE PACKAGE pkg1
IS
    PROCEDURE proc1;
    PROCEDURE proc2;
END;
/
CREATE OR REPLACE PACKAGE BODY pkg1
IS
    PROCEDURE proc1
    IS
        PROCEDURE nested_in_proc1
        IS
            BEGIN
                RAISE VALUE_ERROR;
            END;
        BEGIN
            nested_in_proc1;
        END;
    PROCEDURE proc2
    IS
        BEGIN
            proc1;
        EXCEPTION
            WHEN OTHERS THEN RAISE NO_DATA_FOUND;
        END;
    END pkg1;
/
CREATE OR REPLACE PROCEDURE proc3
IS
    BEGIN
        pkg1.proc2;
    END;
/
BEGIN
    proc3;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line (backtrace_to);
END;
```

```
HR.PKG1 line 19
```

Несколько замечаний по поводу использования UTL_CALL_STACK:

- Оптимизации компилятора могут изменять глубину (лексическую, динамическую и глубину трассировки), поскольку процесс оптимизации может привести к пропуску вызовов подпрограмм.
- Не поддерживаются вызовы UTL_CALL_STACK за границами RPC (Remote Procedure Call). Например, если `proc1` вызывает удаленную процедуру `remoteproc2`, то `remoteproc2` не сможет получить информацию о `proc1` с использованием UTL_CALL_STACK.
- Информация о лексических элементах предоставляется средствами условной компиляции PL/SQL, а не через UTL_CALL_STACK.

Пакет UTL_CALL_STACK чрезвычайно удобен, но для повседневного использования вам, скорее всего, придется построить собственные инструменты на базе подпрограмм пакета. Я написал вспомогательный пакет (см. файлы `12c_utl_call_stack_helper.sql` и `12c_utl_call_stack_helper_demo.sql` на сайте книги) с программами, которые, как мне кажется, могут вам пригодиться.

DBMS_APPLICATION_INFO

Встроенный пакет DBMS_APPLICATION_INFO предоставляет API для «регистрации» текущего состояния выполнения приложения в базе данных Oracle. Для внешнего наблюдения за сохраненной информацией о состоянии приложения используются виртуальные таблицы `V$`. Именно использование виртуальных таблиц `V$` в качестве хранилища трассировочных данных отличает этот пакет от других решений.

Регистрация приложений средствами DBMS_APPLICATION_INFO открывает гораздо больше возможностей для анализа приложений на предмет производительности и потребления ресурсов, чем все остальные решения. Это открывает дополнительные возможности для оптимизации приложений и более точной оценки эксплуатационных затрат.

В следующей таблице перечислены подпрограммы этого пакета. Все они являются процедурами и не могут выполняться в коде SQL.

| Имя | Описание |
|---|---|
| DBMS_APPLICATION_INFO.SET_MODULE | Назначает имя выполняемого модуля |
| DBMS_APPLICATION_INFO.SET_ACTION | Назначает действие внутри модуля |
| DBMS_APPLICATION_INFO.READ_MODULE | Получает информацию о модуле и действии для текущего сеанса |
| DBMS_APPLICATION_INFO.SET_CLIENT_INFO | Назначает информацию о клиенте для сеанса |
| DBMS_APPLICATION_INFO.READ_CLIENT_INFO | Читает информацию о клиенте для сеанса |
| DBMS_APPLICATION_INFO.SET_SESSION_LONGOPS | Записывает строку в таблицу LONGOPS (только в версии 8.0) |

Пример использования пакета DBMS_APPLICATION_INFO:

```

/* Файл в Сети: dbms_application_info.sql */
PROCEDURE drop_dept (
    deptno_IN IN employees.department_id%TYPE
    , reassign_deptno_IN IN employees.department_id%TYPE
)
IS
    l_count PLS_INTEGER;
BEGIN
    DBMS_APPLICATION_INFO.SET_MODULE
        (module_name => 'DEPARTMENT FIXES'

```

```

        ,action_name => null);
DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'GET COUNT IN DEPT');

SELECT COUNT(*)
  INTO l_count
  FROM employees
 WHERE department_id = deptno_IN;
DBMS_OUTPUT.PUT_LINE ('Reassigning ' || l_count || ' employees');

IF l_count > 0
THEN
  DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'REASSIGN EMPLOYEES');

  UPDATE employees
    SET department_id = reassign_deptno_IN
    WHERE department_id = deptno_IN;
END IF;

DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'DROP DEPT');

DELETE FROM departments WHERE department_id = deptno_IN;

COMMIT;

DBMS_APPLICATION_INFO.SET_MODULE(null,null);

EXCEPTION
  WHEN OTHERS THEN
    DBMS_APPLICATION_INFO.SET_MODULE(null,null);
END drop_dept;

```

Обратите внимание на три вызова `DBMS_APPLICATION_INFO`, определяющие три этапа процесса удаления. Это позволяет отслеживать состояние приложения с чрезвычайно высоким уровнем детализации.

Проследите за тем, чтобы имя действия описывало текущую транзакцию или логическую единицу работы внутри модуля.

При завершении транзакции следует вызвать процедуру `DBMS_APPLICATION_INFO.SET_ACTION`, передавая `NULL` в качестве значения параметра `action_name`. Это гарантирует, что последующие транзакции, не зарегистрированные средствами `DBMS_APPLICATION_INFO`, не будут ошибочно включены в текущее действие. Например, если программа обрабатывает исключения, обработчик исключения, вероятно, должен сбрасывать информацию о текущем действии.

Трассировка с использованием `opp_trace`

Программа `opp_trace` доступна на сайте 5-го издания этой книги. Она реализует мощный механизм трассировки, позволяющий направить выходные данные на экран или в таблицу (`opp_trace`). Выполните сценарий `opp_trace.sql`, чтобы создать объекты базы данных (для удаления этих объектов используется сценарий `opp_trace_uninstall.sql`).

Читателю разрешается вносить необходимые изменения в код `opp_trace` для использования его в вашей рабочей среде.

Включение трассировки всех вызовов через `opp_trace` API происходит так:

```
opp_trace.turn_on_trace;
```

В следующем вызове `set_tracing` трассировка включается только для контекстов, содержащих строку «balance»:

```
opp_trace.turn_on_trace ('balance');
```

Теперь посмотрим, как совершаются вызовы `opp_trace.trace_activity` в ваших хранимых программах.

Процедура `q$error_manager.trace` почти никогда не вызывается напрямую. Вместо этого ее вызов вкладывается в вызов `q$error_manager.trace_enabled`, как показано в следующем фрагменте:

```
IF opp_trace.trace_is_on
THEN
  opp_trace.trace_activity (
    context_in => 'generate_test_code for program'
    , data_in   => qu_program_qp.name_for_id (l_program_key)
  );
END IF;
```

Такой способ вызова программы `trace` сводит к минимуму затраты на трассировку. Функция `trace_enabled` возвращает одно логическое значение; она не передает фактических аргументов и завершает свою работу эффективно. Если функция возвращает `TRUE`, то база данных Oracle вычисляет все выражения в списке параметров и вызывает процедуру `trace`, которая также проверит, что для этого конкретного контекста трассировка включена.

Если вызвать процедуру `trace` напрямую в коде приложения, то при каждом переходе к этой строке исполнительное ядро будет вычислять все фактические аргументы в списке параметров и вызывать процедуру `trace`, которая будет проверять, что трассировка включена для этого контекста. Если трассировка отключена, то ничего не происходит, — но приложение уже затратило процессорное время на вычисление аргументов и их передачу `trace`.

Заметит ли пользователь лишние затраты на избыточное вычисление аргументов? Вероятно, не заметит, но с добавлением новых трассировочных вызовов повышается вероятность того, что они повлияют на ощущения пользователя от работы с приложением. Лучше заведите привычку всегда скрывать трассировочные вызовы в команде `IF`, которая сводит лишние затраты к минимуму. Я использую специальный фрагмент кода, чтобы вставка команды `IF` с шаблонным вызовом не требовала ввода большого количества символов.

Пакет DBMS_TRACE

Встроенный пакет `DBMS_TRACE` предоставляет средства для запуска и остановки трассировки PL/SQL в ходе сеанса. При включенной трассировке ядро собирает данные в ходе выполнения программы. Собранная информация выводится в файл на сервере.

Файл, полученный в результате трассировки, содержит информацию о конкретных шагах, выполняемых вашим кодом. `DBMS_PROFILER` и `DBMS_HPROF` (иерархический профайлер), о которых будет рассказано в главе 21, позволяют провести более полный анализ вашего приложения с получением хронометражной информации и данных о количестве выполнений конкретной строки.

Установка пакета DBMS_TRACE

Этот пакет не всегда автоматически устанавливается вместе со всеми остальными встроенными пакетами. Чтобы определить, установлен ли он в вашей системе, подключитесь к базе данных в качестве пользователя с учетной записью `SYS` (или другой учетной записью с привилегиями `SYSDBA`) и выполните в `SQL*Plus` команду:

```
DESCRIBE DBMS_TRACE
```

Если на экране появится сообщение об ошибке:

```
ORA-04043: object dbms_trace does not exist
```

значит, пакет нужно установить. Для этого, используя ту же учетную запись, выполните следующие файлы в указанном порядке:

```
$ORACLE_HOME/rdbms/admin/dbmspbt.sql
$ORACLE_HOME/rdbms/admin/prvtpbt.plb
```

Программы пакета DBMS_TRACE

Пакет DBMS_TRACE содержит три программы.

| Имя | Описание |
|---------------------|--|
| SET_PLSQL_TRACE | Активизирует трассировку в текущем сеансе |
| CLEAR_PLSQL_TRACE | Останавливает трассировку в текущем сеансе |
| PLSQL_TRACE_VERSION | Возвращает основной и дополнительный номера версий пакета DBMS_TRACE |

Для трассировки кода PL/SQL в текущем сеансе выполняется следующий вызов:

```
DBMS_TRACE.SET_PLSQL_TRACE (уровень_трассировки INTEGER);
```

Здесь значением аргумента *уровень_трассировки* служит одна из следующих констант:

- Константы, определяющие, какие элементы программы PL/SQL подлежат трассировке:

```
DBMS_TRACE.trace_all_calls      constant INTEGER := 1;
DBMS_TRACE.trace_enabled_calls  constant INTEGER := 2;
DBMS_TRACE.trace_all_exceptions constant INTEGER := 4;
DBMS_TRACE.trace_enabled_exceptions constant INTEGER := 8;
DBMS_TRACE.trace_all_sql        constant INTEGER := 32;
DBMS_TRACE.trace_enabled_sql    constant INTEGER := 64;
DBMS_TRACE.trace_all_lines      constant INTEGER := 128;
DBMS_TRACE.trace_enabled_lines  constant INTEGER := 256;
```

- Константы, управляющие процессом трассировки:

```
DBMS_TRACE.trace_stop      constant INTEGER := 16384;
DBMS_TRACE.trace_pause     constant INTEGER := 4096;
DBMS_TRACE.trace_resume    constant INTEGER := 8192;
DBMS_TRACE.trace_limit     constant INTEGER := 16;
```



Комбинации констант из пакета DBMS_TRACE активируют одновременную трассировку нескольких элементов языка PL/SQL. Учтите, что константы, управляющие процессом трассировки (например, DBMS_TRACE.trace_pause), не должны использоваться в сочетании с другими константами (такими, как DBMS_TRACE.trace_enabled_calls).

Трассировка всех программ, выполняемых в текущем сеансе, активируется следующим вызовом:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_calls);
```

Трассировка всех исключений, инициируемых в течение текущего сеанса, активируется следующим образом:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_exceptions);
```

Далее запускается программный код. Трассировка прекращается вызовом

```
DBMS_TRACE.CLEAR_PLSQL_TRACE;
```

После этого можно проанализировать содержимое файла трассировки. Имена файлов генерируются Oracle; нужный файл легко находится по дате модификации. О том, где эти файлы хранятся, рассказывается далее, в разделе «Формат собранных данных». Вы также можете установить идентификатор в файле трассировки, чтобы упростить его поиск в будущем:

```
SQL> alter session set tracefile_identifier = 'hello_steven!';
Session altered.
SQL> select tracefile from v$sqlprocess where tracefile like '%hello_steven!%';
TRACEFILE
```

```
-----
/u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_24446_hello_steven!.trc
```

Учтите, что трассировка PL/SQL не может использоваться на серверах коллективного доступа (ранее называвшихся *многопоточными серверами*, MTS).

Управление содержимым файла трассировки

Файлы, генерируемые пакетом DBMS_TRACE, могут быть *очень* большими. Чтобы уменьшить объем выходных данных, следует трассировать не все подряд, а только конкретные программы (этот прием не может использоваться с удаленными вызовами RPC).

Чтобы включить трассировку всех программ, созданных или измененных в ходе сеанса, измените настройки сеанса следующей командой:

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
```

Если вы не хотите менять конфигурацию всего сеанса, перекомпилируйте конкретный программный модуль в режиме отладки следующим образом (способ неприменим к анонимным блокам):

```
ALTER [PROCEDURE | FUNCTION | PACKAGE BODY] имя_программы COMPILE DEBUG;
```

Затем иницилируйте трассировку только этих программных модулей:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_calls);
```

Трассировку также можно ограничить только теми исключениями, которые иницилируются в доступных для данного процесса программах:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_exceptions);
```

Если вы установили два уровня трассировки (для всех программ или исключений, а также для некоторых программ и исключений), будет действовать уровень «для всех».

Приостановление и возобновление трассировки

Процедура SET_PLSQL_TRACE способна не только определять информацию, подлежащую трассировке. С ее помощью также можно приостанавливать и возобновлять процесс трассировки. Например, следующая команда приостанавливает сбор данных вплоть до последующего возобновления трассировки:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_pause);
```

Пакет DBMS_TRACE включает в файл трассировки записи с информацией о приостановлении и возобновлении трассировки.

При помощи константы DBMS_TRACE.trace_limit можно указать, что в файле трассировки должна сохраняться информация только о 8192 трассируемых событиях. Этот подход защищает базу данных от переполнения трассировочной информацией. По завершении сеанса трассировки будут сохранены лишь 8192 последние записи.

Формат собранных данных

Если вы запросили трассировку только для определенных программных модулей, но не текущей программы, данные трассировки сохраняться не будут. Если же включена трассировка текущей программы, в файл трассировки записываются ее тип, имя и глубина стека.

При трассировке исключений определяется номер строки программы, в процессе выполнения которой оно было инициировано, его тип (пользовательское или

предопределенное) и в случае предопределенного исключения — его номер (номер ошибки, связанной с пользовательским исключением, всегда равен 1).

Пример выходных данных трассировки для процедуры `showemps`:

```
*** 1999.06.14.09.59.25.394
*** SESSION ID:(9.7) 1999.06.14.09.59.25.344
----- PL/SQL TRACE INFORMATION -----
Levels set : 1
Trace: ANONYMOUS BLOCK: Stack depth = 1
Trace: PROCEDURE SCOTT.SHOWEMPS: Call to entry at line 5 Stack depth = 2
Trace: PACKAGE BODY SYS.DBMS_SQL: Call to entry at line 1 Stack depth = 3
Trace: PACKAGE BODY SYS.DBMS_SQL: Call to entry at line 1 Stack depth = 4
Trace: PACKAGE BODY SYS.DBMS_SQL: ICD vector index = 21 Stack depth = 4
Trace: PACKAGE PLVPRO.P: Call to entry at line 26 Stack depth = 3
Trace: PACKAGE PLVPRO.P: ICD vector index = 6 Stack depth = 3
Trace: PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace: PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace: PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 4
```

Отладка программ PL/SQL

Тестирование программы — это поиск ошибок в программном коде, а ее отладка — это определение и устранение причин, обусловивших их появление. Это два совершенно разных процесса, которые не следует путать. Протестировав программу и выявив ошибки, разработчик должен их исправить. С этого момента и начинается отладка.

Многие программисты считают отладку самой трудной частью программирования. Причины ее сложности часто заключаются в следующем:

- **Недостаточное понимание задачи, решаемой программой.** Большинство разработчиков предпочитают заниматься лишь написанием программного кода и не любят тратить время на то, чтобы разбираться в спецификации. Вероятность того, что написанная таким образом программа будет соответствовать требованиям, ничтожно мала.
- **Плохой стиль программирования.** Трудные для чтения программы (плохо документированные, содержащие слишком обширную документацию, плохо отформатированные, с неудачными именами переменных и т. п.), а также программы, неправильно разделенные на модули, отлаживать гораздо сложнее, чем хорошо спроектированный и структурированный код.
- **Большое количество ошибок в программе.** Если разработчик не имеет необходимых навыков анализа и программирования, скорее всего, его код будет содержать множество ошибок. Без четко определенного, пошагового метода исправления ошибок вы, скорее всего, не устоите под их напором.
- **Слабые навыки отладки.** Существует много разных подходов к выявлению ошибок. Некоторые из них только затрудняют работу, поэтому неумение правильно выбрать технологию отладки оборачивается лишними затратами времени, головной болью и недовольством начальства.

В следующих разделах рассматриваются методы отладки, которыми лучше не пользоваться, а также методы, способные значительно облегчить этот процесс и сделать его более эффективным.

Неправильные способы организации отладки

Вероятно, при виде приведенных ниже описаний почти каждый читатель скажет себе: «Ну разумеется, это же очевидно. Конечно, так поступать нельзя. Я никогда такого не сделаю».

И все же в следующий раз, сев за работу, многие из вас запросто могут последовать этим, таким очевидно неправильным методам отладки.

Надеюсь, если вы хотя бы отчасти узнаете себя в этих описаниях, это поможет вам изменить свой подход к делу.

Хаотичная отладка

Столкнувшись с ошибкой, вы начинаете бурную деятельность по поиску причины ее возникновения. И хотя сам факт ошибки может свидетельствовать о том, что задача плохо проанализирована или что не найден наиболее удачный метод ее решения, вы не пытаетесь разобраться в программе. Вместо этого вы включаете в программу множество команд MESSAGE (в Oracle Forms), SRW.MESSAGE (в Oracle Reports) или DBMS_OUTPUT.PUT_LINE (в хранимых модулях) и надеетесь, что это поможет.

Причем вы даже не сохраняете копию программы перед внесением изменений, чтобы не тратить лишнее время. Вам, как обычно, некогда, поэтому вы уверены, что ответы на все вопросы появятся сами собой, после чего можно будет удалить отладочные команды.

В результате масса времени тратится на просмотр ненужной информации. Вы проверяете все подряд, хотя большинство используемых конструкций давно выверены и отлажены.

Вы пропускаете обед, чтобы не отвлекаться от работы, поглощаете кофе, пытаетесь максимально сконцентрировать свое внимание. Даже не имея ни малейшего понятия о причине возникновения проблемы, вы думаете, что если попробовать то-то и то-то, это может помочь. Вы вносите изменения, тратите несколько минут на компиляцию, написание и исполнение теста — и убеждаетесь в бесплодности всех усилий. В результате появляются новые проблемы, поскольку вы не учли влияния вносимых изменений на другие части программы.

В итоге вы возвращаете программу в исходное состояние и пытаетесь сделать что-нибудь еще. Ваш коллега, заметив, что у вас уже дрожат руки, предлагает свою помощь. Однако вы не знаете, как описать проблему, и у вас даже нет исходной версии, которую можно было бы показать, — программа превратилась в минное поле, сплошь усеянное командами трассировки. В итоге вы отвергаете помощь лучшего программиста в группе, звоните домой и сообщаете, что к ужину вы сегодня не придете.

Почему? Потому, что нужно исправить ошибку!

Иррациональная отладка

Вы сгенерировали отчет, а он оказался пустым. При этом вы уверены, что вашей вины здесь нет, хотя в течение последнего часа занимались тем, что вносили изменения в структуры данных и программный код для их запроса и форматирования.

Вы звоните в отдел внутренней поддержки и спрашиваете, нет ли проблем с сетью, хотя File Manager показывает, что все сетевые диски доступны. Потом вы интересуетесь, не дедалась ли куда-нибудь база данных, хотя только что успешно к ней подключались, отнимаете 10 минут времени у специалиста из отдела поддержки и, ничего не добившись, в раздражении вешаете трубку.

«Они там ничего не понимают», — бормочете вы. Но как бы там ни было, источник проблемы придется искать самостоятельно. Вы углубляетесь в только что измененный код и проверяете каждую строку. Следующие два часа вы вслух разговариваете с самим собой: «А это что? Я вызвал хранимую процедуру в команде IF. Раньше я этого никогда не делал. Может, хранимые программы так вызывать нельзя?» Вы удаляете команду IF, заменяя ее командой GOTO, но это не решает проблему.

Поиски продолжаются: «Вроде бы код правильный. Но он вызывает программу, которую когда-то написал Джо». Джо уже давно уволился, поэтому, конечно же, виноват он:

«Программа, наверное, уже не работает, тем более что мы обновили систему голосовой почты». Вы решаете протестировать программу Джо, которая не изменялась уже два года. Программа работает... только не в вашем коде.

Вы постепенно приходите в отчаяние. «Может, этот отчет должен выполняться только на выходных? А если разместить локальный модуль в анонимном блоке? И вроде я что-то слышал об ошибках в этой служебной программе. Надо поискать обходное решение...»

Вскоре вы начинаете понимать своего восьмилетнего сына, который бьет по монитору, когда не может пройти последний уровень в своей любимой компьютерной игре. И уже собравшись домой, вы вдруг осознаете, что подключены к базе данных, предназначенной для разработки и почти не содержащей никаких данных. Вы подключаетесь к другому экземпляру базы данных, запускаете отчет, и все оказывается в порядке.

За исключением того, что ваш отчет теперь содержит лишние команды `GO` и массу других добавленных конструкций...

Полезные советы и стратегии отладки

Эта глава не претендует на звание исчерпывающего руководства по отладке. Однако приведенные здесь советы и приемы помогут вам улучшить навыки исправления ошибок.

Пользуйтесь отладчиком исходного кода

Самый эффективный способ сокращения времени отладки кода — использование отладчика. Отладчик присутствует в практически любой интегрированной среде разработки приложений (IDE) PL/SQL. И если вы пользуетесь SQL Navigator или Toad от Quest, PL/SQL Developer от Allround Automations, Oracle SQL Developer или другим сходным инструментом с графическим интерфейсом, то сможете легко устанавливать в программах точки останова, выполнять код в пошаговом режиме, просматривать значения переменных и т. д.

Советы, приведенные в этом разделе, полезны независимо от того, пользуетесь ли вы отладчиком с графическим интерфейсом, но если вы отлаживаете программы старым дедовским методом (вставляя в программный код многочисленные вызовы `DBMS_OUTPUT.PUT_LINE`), эффективность вашей работы оставляет желать лучшего. (К сожалению, если ваш код развернут на площадке клиента, отладка с использованием графических средств не всегда возможна; в таких случаях приходится использовать различные средства протоколирования.)

Собирайте информацию

Соберите как можно больше сведений о том, где, когда и при каких условиях произошла ошибка. Маловероятно, что, столкнувшись с этой ошибкой впервые, вы получите о ней все необходимые данные. И помните, что многие проблемы, которые на первый взгляд кажутся простыми, на самом деле могут потребовать тщательного тестирования и анализа. Так что не бросайтесь сразу же модифицировать код в уверенности, что вы понимаете, в чем здесь дело, а сначала выполните следующие действия:

1. Запустите программу еще раз и посмотрите, воспроизводится ли ошибка. Если ошибка не повторяется, вам вряд ли удастся понять ее причины и исправить ее. Так что постарайтесь определить условия, при которых ошибка повторяется, — это даст вам много полезной информации.
2. Ограничьте область тестирования и отладки программы — возможно, так вы значительно быстрее найдете ошибку. Недавно я отлаживал один из своих модулей Oracle Forms, у которого в одном из всплывающих окон происходила потеря

данных. На первый взгляд закономерность выглядела так: «Если для нового вызова ввести только один запрос, этот запрос будет потерян». Если бы я остановил тестирование в этот момент, мне пришлось бы проанализировать весь код инициализации записи вызова и отработки логики `INSERT`. Вместо этого я опробовал разные варианты ввода данных и вскоре обнаружил, что данные теряются только при переходе к всплывающему окну от конкретного элемента. Тестовый сценарий значительно сузился, и обнаружить ошибку в логике стало совсем несложно.

3. Проанализируйте обстоятельства, при которых ошибка не возникает. Выявление подобных ситуаций помогает сузить область поиска и осознать причину ошибки.

Чем больше информации об ошибке вы соберете, тем легче будет найти и устранить причину ее возникновения. Поэтому не жалейте времени на дополнительное тестирование и анализ поведения программы.

Не теряйте здравый смысл

Символическая логика — хлеб насущный для программистов. Какой бы язык программирования вы ни использовали, все программы строятся по определенным логическим правилам. Язык PL/SQL имеет один синтаксис, язык C — другой. В них применяются разные ключевые слова и команды (хотя есть и общие — например `IF`, но их спецификации несколько различаются). И уж совсем иначе выглядит программа на языке LISP. Однако за всеми этими языками стоит логика, выражаемая с помощью тех или иных операторов.

Именно логическая строгость и облегчает изучение новых языков программирования. Если вы в состоянии четко определить для себя задачу и выработать последовательность ее решения, особенности конкретного языка вторичны.

И все же программисты удивительным образом ухитряются обходить всякую логику и здравый смысл и выбирают самые иррациональные пути решения проблем. Как только что-то идет не так, мы начинаем принимать желаемое за действительное, подозревать и обвинять всех и все вокруг: коллег, компьютер, компилятор, текстовый редактор... кого угодно, только не себя.

Но помните, что не желая признать свой промах, вы отказываетесь от поиска путей решения проблемы. Компьютеры и компиляторы не отличаются интеллектом, но они быстры, надежны и последовательны. Все, что они умеют, — это следовать правилам, записанным в вашей программе. Так что, обнаружив ошибку в программном коде, примите ответственность за нее. Признайте, что именно вы сделали что-то не так — вы, а не компилятор PL/SQL, не Oracle Forms и не текстовый редактор.

Если вы начинаете ставить под сомнение базовые элементы или правила компилятора, которые всегда работали в прошлом, лучше отдохнуть от работы. А еще лучше — показать ваш код кому-нибудь другому. Просто удивительно, как сторонний взгляд порой помогает направить ваши аналитические способности на истинную причину проблемы.



Будьте рациональны и бесстрастны. Соглашайтесь с тем, что логично. Отвергайте то, что не имеет объяснения.

Анализ вместо поспешных действий

Итак, когда вы собрали всю необходимую информацию об ошибке, ее нужно проанализировать. У многих программистов анализ выглядит так: «Хм, наверное, причина в этом. Сейчас внесу изменения, перекомпилирую и посмотрю, что получится».

Чем плох подобный подход? Если вы ищете решение методом проб и ошибок, это означает, что:

- вы не уверены в том, действительно ли данное изменение способно решить проблему; имея такую уверенность, вы бы не «проверяли», а просто тестировали внесенное изменение;
- вы не проанализировали ошибку, чтобы найти причины ее появления; если бы вы имели представление о том, чем вызвана ошибка, то знали бы, как ее исправить; не зная причин, вы пытаетесь что-то наобум изменить и проанализировать результат; это в высшей степени порочная логика;
- даже если внесенное вами изменение устраняет ошибку, вы не знаете наверняка, не приняла ли она другую форму и не появится ли вновь (если вы не понимаете суть проблемы, ее внешнее исчезновение еще не означает, что она не появится снова); все, что можно сказать в данном случае, — это то, что при известных вам условиях данная проблема больше не проявляется.

Чтобы действительно избавиться от проблемы, ее нужно полностью проанализировать и определить ее источник. Только тогда будет найдено правильное и надежное решение.

Отыскав потенциальное решение, очень внимательно проанализируйте программный код, не выполняя его. Попробуйте придумать разные сценарии для проверки своей гипотезы. И лишь обретя уверенность в том, что вы действительно поняли проблему и нашли ее решение, внесите изменения и протестируйте новую версию программы. Помните: нужно не *пробовать*, а *исправлять* и *тестировать*.

Проанализируйте ошибку, прежде чем тестировать решения. Если вы говорите себе: «А почему бы не попробовать так?» в надежде, что это решит проблему, вы лишь напрасно тратите время, а отладка проходит неэффективно.

Делайте перерывы и обращайтесь за помощью

В том, что касается решения проблем — будь то ошибка в программе или личный кризис, — самым слабым звеном часто оказывается сам решающий. При слишком сильном погружении в проблему трудно сохранить способность оценивать ее объективно.

Если вы чувствуете, что застряли и не продвигаетесь вперед, если все средства перепробованы, а решения все нет, примите радикальные меры:

1. Сделайте перерыв.
2. Обратитесь за помощью.

Если накопились усталость и психологическое напряжение, если вы потеряли цель и начали действовать нерационально, если вы уже много часов не отрываете взгляда от экрана, попробуйте отвлечься и немного передохнуть — возможно, решение появится само собой.

Мой упадок духа нередко напрямую связан с тем временем, которое я провожу в своем эргономичном кресле, положив руки на клавиатуру с подкладками для кистей и уставившись в монитор с пониженным излучением. Часто достаточно отойти от компьютера, чтобы голова прояснилась и в ней забрезжило готовое решение. Вы никогда не просыпались утром после очень трудного рабочего дня, когда внезапно неувловимый ответ приходит сам собой?

Возьмите себе за правило хотя бы один раз в час вставать из-за стола, чтобы немного прогуляться. Расслабьтесь и позвольте нейросетям вашего мозга установить новые соединения — и возможно, у вас появятся новые идеи. Окружающий мир продолжает существовать, даже когда ваши глаза прикованы к экрану. Взгляните, что происходит

за окном. Всегда полезно помнить об этом — хотя бы для того, чтобы увидеть, какая погода за пределами вашего кокона с кондиционером.

Еще более эффективный путь — попросить кого-нибудь просмотреть ваш код. Взгляд со стороны порой творит чудеса. Можно часами сражаться с программой, а потом, рассказывая кому-нибудь о своих затруднениях, вдруг понять, в чем дело. Ошибка может быть очень простой: например, это может быть несоответствие имен, неверное предположение или неправильная логика оператора `IF`. Но даже если на вас не снизойдет озарение, причину ошибки поможет выявить свежий взгляд постороннего человека, который:

- не писал этот код и не имеет неверных предположений о том, как он должен работать;
- не заиклен на программе и вообще не беспокоится о том, будет она работать или нет.

Кроме того, обращаясь за помощью к коллегам, вы демонстрируете им, что уважаете их мнение и квалификацию. Это помогает создать атмосферу доверия, взаимопомощи и сотрудничества, что в конечном счете повышает эффективность работы всей группы.

Изменяйте и тестируйте разные области кода по очереди

Один из главных недостатков хорошего программиста заключается в том, что он слишком уверен в своих способностях и поэтому пытается решать одним махом множество проблем. Внеся пять–десять изменений и запустив тест, мы получаем ненадежные и практически бесполезные результаты. Во-первых, некоторые изменения порождают новые проблемы (весьма распространенное явление и верный признак того, что программа требует дополнительной отладки и тестирования); во-вторых, исчезают не все исходные ошибки, и мы понятия не имеем, какие изменения способствовали устранению ошибок, а какие послужили причиной появления новых.

Короче говоря, вся работа по отладке лишь вносит путаницу, поэтому приходится вернуться к исходной точке и действовать более последовательно.

Если только вы не ограничиваетесь очень простыми изменениями, старайтесь решать проблемы по одной, а затем тестировать решения. Возможно, это увеличит затраты времени на компиляцию, генерирование и тестирование кода, но в долгосрочной перспективе ваша работа станет более производительной.

Другой аспект пошагового тестирования и отладки — проведение *модульного тестирования* отдельных модулей перед тестированием программы, вызывающей эти модули. Если вы тестируете свои программы по отдельности и определяете, что они работают, то при отладке приложения в целом (общесистемный тест) вам не придется беспокоиться о том, возвращают ли эти модули правильные значения и выполняют ли они правильные действия. Вместо этого можно сконцентрироваться на коде, который вызывает модули (за дополнительной информацией о модульном тестировании обращайтесь к разделу «Тестирование программ PL/SQL»).

Также будет полезно найти систему для хранения информации о ваших усилиях по диагностике ошибок. Дэн Кламадж, рецензент этой книги, сообщает, что он ведет простой текстовый файл с комментариями о его усилиях по воспроизведению проблемы и о том, что было сделано для ее исправления. Обычно в этот файл включается весь код SQL, написанный для анализа ситуации, конфигурационные данные для тестовых сценариев, список проанализированных модулей и все остальные данные, которые могут представлять интерес в будущем. С таким файлом намного проще вернуться к проблеме в будущем (например, когда вы отоспитесь и будете снова готовы к работе) и воспроизвести исходную цепочку рассуждений.

«Белые списки» и управление доступом к программным модулям

Приложения PL/SQL обычно строятся из пакетов, одни из которых относятся к API верхнего уровня, используемого программистами для реализации пользовательских требований, а другие решают вспомогательные задачи и предназначены для использования только другими пакетами.

До выхода Oracle Database 12c язык PL/SQL не мог помешать сеансу использовать любые подпрограммы пакетов, для которых схема данного сеанса обладает привилегией EXECUTE.

В Oracle Database 12c все программные модули PL/SQL имеют необязательную секцию ACCESSIBLE BY для определения «белого списка» модулей PL/SQL, которые могут обращаться к создаваемым или изменяемым модулям PL/SQL. Рассмотрим простой пример. Я начинаю с создания спецификации своего «общедоступного» пакета, который должен использоваться другими разработчиками для построения приложения:

```
/* Файл в Сети: 12c_accessible_by.sql */
CREATE OR REPLACE PACKAGE public_pkg
IS
    PROCEDURE do_only_this;
END;
/
```

Затем я создаю спецификацию «приватного» пакета — «приватного» в том смысле, что он должен вызываться только из общедоступного пакета. Соответственно, я добавляю секцию ACCESSIBLE_BY:

```
CREATE OR REPLACE PACKAGE private_pkg
    ACCESSIBLE BY (public_pkg)
IS
    PROCEDURE do_this;
    PROCEDURE do_that;
END;
/
```

Далее можно переходить к реализации тел пакетов. Процедура public_pkg.do_only_this вызывает подпрограммы приватного пакета:

```
CREATE OR REPLACE PACKAGE BODY public_pkg
IS
    PROCEDURE do_only_this
    IS
    BEGIN
        private_pkg.do_this;
        private_pkg.do_that;
    END;
END;
/
CREATE OR REPLACE PACKAGE BODY private_pkg
IS
    PROCEDURE do_this
    IS
    BEGIN
        DBMS_OUTPUT.put_line ('THIS');
    END;

    PROCEDURE do_that
    IS
    BEGIN
        DBMS_OUTPUT.put_line ('THAT');
    END;
END;
/
```

Процедура общедоступного пакета запускается без малейших проблем:

```
BEGIN
    public_pkg.do_only_this;
END;
/
THIS
THAT
```

Но при попытке вызвать подпрограмму приватного пакета в анонимном блоке я получу сообщение об ошибке:

```
BEGIN
    private_pkg.do_this;
END;
/
```

```
ERROR at line 2:
ORA-06550: line 2, column 1:
PLS-00904: insufficient privilege to access object PRIVATE_PKG
ORA-06550: line 2, column 1:
PL/SQL: Statement ignored
```

Та же ошибка происходит при попытке откомпилировать программный модуль, пытающийся вызвать подпрограмму из приватного пакета:

```
SQL> CREATE OR REPLACE PROCEDURE use_private
2  IS
3  BEGIN
4      private_pkg.do_this;
5  END;
6  /
```

Warning: Procedure created with compilation errors.

```
SQL> sho err
Errors for PROCEDURE USE_PRIVATE:
LINE/COL ERROR
-----
4/4      PL/SQL: Statement ignored
4/4      PLS-00904: insufficient privilege to access object PRIVATE_PKG
```

Как видно из текста сообщений, проблема обнаруживается во время компиляции. Использование данной возможности не приводит ни к каким отрицательным последствиям во время выполнения.

Защита хранимого кода

Почти каждое приложение содержит закрытую информацию. Например, автор коммерческого приложения PL/SQL не захочет, чтобы пользователи (или конкуренты) видели все их секреты. Oracle предоставляет специальную программу `wrap`, предназначенную для сокрытия секретов приложения.



Некоторые разработчики считают, что программа `wrap` «шифрует» код, но это неверно. Если вам нужно передать действительно секретную информацию (например, пароль), полагаться на эту программу не стоит. Однако следует иметь в виду, что с помощью встроенного пакета `DBMS_CRYPTO` (или `DBMS_OBFUSCATION_TOOLKIT` до выхода Oracle10g) Oracle позволяет реализовать в приложении полноценную защиту данных. Шифрование и другие аспекты безопасности приложений PL/SQL описаны в главе 23.

Программа `wrap` преобразует ASCII-текст исходного кода в нечитаемую форму, которую можно передавать пользователям, в региональные филиалы и т. д. для создания соответствующих программ в других экземплярах базы данных. Код характеризуется такой же переносимостью, что и исходный код PL/SQL, и может импортироваться и экспортироваться. Oracle поддерживает зависимости для программ со скрытым кодом точно так же, как для программ с обычным кодом. Иначе говоря, скрытые программы в базе данных с функциональной точки зрения ничем не отличаются от обычных программ PL/SQL; единственное различие заключается в том, что их нельзя прочитать через представление `USER_SOURCE`.

Исполняемая программа `wrap` существует уже много лет. Начиная с Oracle10g Release 2 для сокрытия динамически сконструированного кода PL/SQL применяются программы `DBMS_DDL.WRAP` и `DBMS_DDL.CREATE_WRAPPED`.

Ограничения

При работе со скрытым кодом необходимо учитывать следующие обстоятельства:

- Сокрытие усложняет восстановление исходного кода, однако пароли и другие конфиденциальные данные не должны храниться в коде (и помните, что обработанный код *можно* восстановить в доступной для человека форме).
- Исходный код триггеров скрываться не может. Если вам необходимо скрыть содержимое триггеров, переместите код в отдельный пакет, а затем вызовите соответствующую программу из триггера.
- Скрытый код не может компилироваться в экземплярах Oracle версий ниже той, из которой была взята программа `wrap`.
- В коде, к которому будет применяться сокрытие, не могут использоваться подстановочные переменные SQL*Plus.

Использование программы `wrap`

Чтобы скрыть исходный код PL/SQL, нужно запустить исполняемый файл `wrap.exe` из каталога `bin` экземпляра Oracle. Формат команды:

```
wrap iname=исх_файл [oname=рез_файл]
```

Здесь *исх_файл* — исходная версия программы, а *рез_файл* — выходной файл для записи скрытой версии кода. Если исходный файл задан без расширения, по умолчанию используется расширение `.sql`.

Если аргумент *рез_файл* не задан, программа `wrap` создает файл с именем, совпадающим с именем исходного файла, и расширением `.plb`.

Несколько примеров использования программы `wrap`:

- Сокрытие программы с параметрами по умолчанию:

```
wrap iname=secretprog
```
- Сокрытие тела пакета с переопределением всех параметров. Обратите внимание: скрытый файл не обязательно должен иметь то же имя или расширение, что и исходный:

```
wrap iname=secretbody.spb oname=shhhhhh.bin
```

Динамическое сокрытие кода с использованием `DBMS_DDL`

В Oracle10g Release 2 появился механизм сокрытия динамически генерируемого кода: программы `WRAP` и `CREATE_WRAPPED` из пакета `DBMS_DDL`:

- DBMS_DDL.WRAP — возвращает строку, содержащую скрытую версию кода.
 - DBMS_DDL.CREATE_WRAPPED — компилирует скрытую версию кода в базу данных.
- Обе эти программы имеют перегруженные версии, работающие с одной строкой и с массивами строк на базе коллекционных типов DBMS_SQL.VARCHAR2A и DBMS_SQL.VARCHAR2S. Пара примеров использования этих программ:

- Соккрытие строки, создающей небольшую процедуру, и ее вывод:

```
SQL> DECLARE
2     l_program    VARCHAR2 (32767);
3 BEGIN
4     l_program := 'CREATE OR REPLACE PROCEDURE dont_look IS BEGIN NULL; END;';
5     DBMS_OUTPUT.put_line (SYS.DBMS_DDL.wrap (l_program));
6 END;
7 /
```

Результат выполнения:

```
CREATE OR REPLACE PROCEDURE dont_look wrapped
a0000000
369
abcd
....
XtQ19En0I8a6hBSJmk2NebMgPHswg5nnm7+fMr2ywFy4CP6Z9P4I/v4rpXQruMAy/tJepZmB
CC0r
uIHHLcmmpkOCnm4=
```

- Чтение определения программы PL/SQL из файла, соккрытие кода и его компиляция в базу данных:

```
/* Файл в Сети: obfuscate_from_file.sql */
PROCEDURE obfuscate_from_file (
    dir_in    IN    VARCHAR2
    , file_in  IN    VARCHAR2
)
IS
    l_file     UTL_FILE.file_type;
    l_lines    DBMS_SQL.varchar2s;

    PROCEDURE read_file (lines_out IN OUT NOCOPY DBMS_SQL.varchar2s)
    IS BEGIN ... .. END read_file;
BEGIN
    read_file (l_lines);
    SYS.DBMS_DDL.create_wrapped (l_lines, l_lines.FIRST, l_lines.LAST);
END obfuscate_from_file;
```

Работа со скрытым кодом

Ниже приведено несколько советов по работе со скрытым кодом.

- Создайте командные файлы для быстрого и унифицированного сокращения файлов. Например, в Windows NT в каталоге исходных файлов можно создать файл с расширением .bat и включить в него следующую строку:

```
c:\orant\bin\wrap iname=plvrep.sps oname=plvrep.pls
```

Конечно, также можно создать параметризованный сценарий и передать ему имена скрываемых файлов.

- Скрывать можно только спецификации и тела пакетов, спецификации и тела объектов типов, независимые функции и процедуры. Если программе wrap передать файл, содержащий другой код SQL или PL/SQL, этот файл изменен не будет.
- Чтобы узнать, скрыта ли программа, достаточно посмотреть на ее заголовок. Если программа скрыта, заголовок будет содержать ключевое слово WRAPPED:

```
PACKAGE BODY имя_пакета WRAPPED
```


Впрочем, даже если вы не заметите указанного слова в заголовке, то сразу поймете, что исходный код скрыт, потому что в представлении USER_SOURCE он будет выглядеть примерно так:

```
LINE TEXT
```

```
-----  
45 abcd  
46 95a425ff  
47 a2  
48 7 PACKAGE:
```

- Скрытый код имеет намного больший объем, чем исходный. Так, пакет размером 57 Кбайт после сокрытия занял 153 Кбайт, а пакет размером 86 Кбайт увеличился до 357 Кбайт. Таким образом, для хранения скрытого кода в базе данных требуется больше места. Размер откомпилированного кода остается неизменным, хотя время компиляции тоже может увеличиться.

Знакомство с оперативной заменой (Oracle11g Release 2)

Одним из самых значительных усовершенствований Oracle11g Release 2 является *оперативная замена* — новый элемент технологий высокой доступности Oracle. Эта функция позволяет заменять компоненты баз данных во время использования приложения; иначе говоря, Oracle позволяет изменять приложения PL/SQL на ходу. Оперативная замена сводит к минимуму (а то и полностью устраняет) простои в работе приложения.

Когда возникает необходимость в обновлении используемого приложения, Oracle создает копию соответствующих объектов базы данных, используемых приложением, и *переопределяет* скопированные объекты в изоляции от работающего приложения. Вносимые изменения никак не проявляются и не оказывают воздействия на пользователей. Пользователь продолжает работать с приложением в том виде, в котором оно существовало до внесения изменений (то есть до перехода на новую версию). Когда вы будете убеждены в том, что все изменения верны, доступ к обновленному приложению открывается всем пользователям.

Как нетрудно предположить, добавление этой возможности оказало колоссальное влияние на Oracle. Например, если вы хотите получить список *всех* определенных вами объектов, вместо запроса к ALL_OBJECTS следует запросить содержимое ALL_OBJECTS_AE (AE=«All Editions»). Теперь уникальный спецификатор объекта образуется из значений OWNER, OBJECT_NAME и EDITION_NAME (если предположить, что для владельца включена поддержка переопределения). Этот аспект дает всего лишь отдаленное представление об изменениях, которые оперативное переопределение породило в Oracle.

Другие возможности Oracle, относящиеся к области высокой доступности, могут применяться таким образом, что установка приложения не требует никакой специальной подготовки, а разработчики даже не знают, какие средства высокой доступности используются на текущей площадке.

С оперативной заменой дело обстоит иначе. Ее использование возможно лишь при соблюдении следующих условий:

- Подготовка приложения к оперативной замене требует изменения схем, которым принадлежат объекты базы данных. Эта работа выполняется архитектором приложения, а ее результаты отражаются в новой (или самой первой) версии приложения. Для реализации этого подготовительного шага пишутся специальные сценарии, которые должны запускаться «традиционно» (то есть в автономном режиме).


```
BEGIN
  RETURN (last_in || ', ' || first_in);
END full_name;
/
```

При выполнении функции мы получаем новый результат:

```
SQL> BEGIN  DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Feuerstein, Steven
```

Но если вернуться к базовой версии, имя снова выводится в старом формате:

```
SQL> ALTER SESSION SET edition = ora$base/
2 BEGIN  DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Steven Feuerstein
```

Конечно, мы рассмотрели простейший пример использования оперативной замены. Проектировщик архитектуры приложения и группа разработки должны подробно разобраться во многих аспектах этого механизма, особенно в триггерах и представлениях, необходимых при изменении структуры таблицы (которая *не поддерживает* переопределение напрямую).

Оперативная замена подробно документирована в документации Oracle11g Release 2 Advanced Application Developer's Guide.

21

Оптимизация приложений PL/SQL

Оптимизация приложения Oracle — это сложный процесс, включающий оптимизацию команды SQL, проверку правильности конфигурации системной глобальной области (SGA), оптимизацию алгоритмов и т. д. Настройка отдельных программ PL/SQL немного проще, но тоже сопряжена с преодолением ряда трудностей. Прежде чем тратить время на попытки повысить производительность кода PL/SQL, необходимо сделать следующее:

- **Настроить доступ к коду и данным в SGA.** Перед выполнением программный код загружается в область памяти Oracle, называемую SGA. Этот процесс может потребовать оптимизации, как правило, осуществляемой администратором базы данных. О SGA и других аспектах архитектуры PL/SQL подробно рассказано в главе 24.
- **Оптимизировать команды SQL.** Практически в любом приложении, разработанном для РСУБД Oracle, большая часть операций по настройке связана с оптимизацией команд SQL. Например, если в программе производится неэффективное объединение данных из 16 таблиц, это сводит на нет все попытки усовершенствования процедурного блока кода. Существует множество инструментальных средств сторонних производителей, которые упрощают работу администраторов баз данных и разработчиков, выполняют сложный анализ SQL-кода и предлагают более эффективные альтернативы.
- **Используйте самый высокий уровень оптимизации.** В Oracle10g появился оптимизирующий компилятор для кода PL/SQL. Используемый по умолчанию уровень 2 прилагал максимум усилий к тому, чтобы ускорить работу вашего кода (Oracle11g поддерживает еще более высокий уровень оптимизации 3; впрочем, по умолчанию продолжает использоваться уровень 2, которого должно быть достаточно для подавляющего большинства ваших программ).

Если вы убедились в том, что контекст выполнения кода PL/SQL не имеет очевидных недостатков из области эффективности, обратите внимание на пакеты и другой код.

- **Используйте при разработке приложения передовые методы и стандарты.** При реализации исходных требований к программе старайтесь применять наиболее эффективные методы, однако не стоит чрезмерно беспокоиться о доведении до совершенства каждой строки кода. Помните, что большая часть написанного кода не является слабым местом программы и не нуждается в оптимизации.
- **Анализируйте эффективность выполнения разных компонентов приложения.** Если скорость работы приложения недостаточно высока, выясните, какие именно элементы приложения замедляют его работу, и сконцентрируйтесь на них.

- **Оптимизируйте алгоритмы.** Поскольку PL/SQL является процедурным языком программирования, его часто применяют для реализации сложных формул и алгоритмов. В программах можно встретить условные операторы, циклы, операторы GOTO и многократно используемые модули. Программные алгоритмы могут быть реализованы множеством разных способов, причем далеко не все из них успешны. Как же оптимизировать неудачно написанный алгоритм? На этот вопрос не существует простого ответа. Ведь процесс оптимизации алгоритмов гораздо сложнее оптимизации команд языка SQL (который является структурированным языком, а следовательно, хорошо подходит для автоматизированного анализа).
- **Пользуйтесь всеми доступными средствами повышения производительности PL/SQL.** Корпорация Oracle включила в свой продукт множество средств, повышающих эффективность выполнения программного кода. Пользуйтесь такими конструкциями, как RETURNING или FORALL, и ни в коем случае не применяйте устаревшие методы, за которые приходится расплачиваться снижением эффективности приложения.
- **Найдите баланс между производительностью и затратами памяти.** Многие приемы, повышающие производительность кода, приводят к дополнительным затратам памяти — обычно PGA (Program Global Area), но иногда и SGA. Молниеносная быстрая не принесет никакой пользы, если затраты памяти окажутся неприемлемыми в вашей рабочей среде.

Подробное рассмотрение оптимизации SQL и настройки базы данных и SGA выходит за рамки книги. Впрочем, в этой книге будут представлены важнейшие средства оптимизации проектировании PL/SQL, а также советы относительно их применения для достижения максимально быстрого выполнения PL/SQL.

Наконец, не стоит забывать, что оптимизация производительности является делом совместным. Работайте в тесном контакте с администратором базы данных, особенно в том, что касается использования таких ключевых возможностей, как коллекции, табличные функции и кэширование результатов функций.

Средства, используемые при оптимизации

В этом разделе будут представлены программные инструменты и методы, применяемые при оптимизации кода. Они делятся на несколько категорий: анализ использования памяти, выявление «узких мест» в коде PL/SQL, хронометраж, выбор самых быстрых программ, предотвращение заикливания и использование предупреждений, относящихся к производительности.

Анализ использования памяти

Как упоминалось ранее, при оптимизации производительности кода также необходимо учитывать объем памяти, занимаемой вашей программой. Данные программы занимают место в PGA; каждый сеанс, подключенный к базе данных Oracle, имеет собственную область PGA. Таким образом, общие затраты памяти приложения обычно многократно превышают объем памяти, необходимой для одного экземпляра программы. Затраты памяти особенно важны при работе с коллекциями, объектными типами с большим количеством атрибутов и записями с большим количеством полей. Эта тема подробнее обсуждается в разделе «PL/SQL и память экземпляров базы данных» в главе 24.

Выявление «узких мест» в коде PL/SQL

Прежде чем заниматься оптимизацией приложения, необходимо разобраться, что работает медленно и на чем следует сосредоточить усилия. Oracle и сторонние разработчики

предлагают разнообразные программные средства, упрощающие решение этой задачи; обычно они анализируют команды SQL в вашем коде, предлагают альтернативные реализации и т. д. Эти инструменты чрезвычайно полезны, но они нередко сбивают с толку разработчиков PL/SQL: выдавая объемистые данные из области производительности, они не отвечают на самый важный вопрос: где находятся «узкие места» в вашем коде?

Для получения этой информации Oracle предлагает ряд своих встроенных средств. Самые полезные из них:

- **DBMS_PROFILER** — встроенный пакет для сбора профильных данных. При запуске кода Oracle собирает в специальных таблицах подробную информацию о том, сколько времени заняло выполнение каждой строки в вашем коде. Далее вы можете обращаться к этим таблицам с запросами или (предпочтительный вариант) используйте такие продукты, как Toad или SQL Navigator, для вывода данных в удобном графическом виде.
- **DBMS_HPROF** — в Oracle11g появился новый иерархический профайлер, способный легко получать данные производительности из стека вызовов. **DBMS_PROFILER** предоставляет «плоские» данные производительности, и при работе с ним трудно получить ответы на вопросы типа: «Сколько времени потрачено на выполнение процедуры **ADD_ITEM?**» Иерархический профайлер упрощает получение подобной информации.

DBMS_PROFILER

На тот случай, если ваш инструментарий не предлагает интерфейс к **DBMS_PROFILER**, ниже приводятся некоторые инструкции и примеры.

Прежде всего следует отметить, что Oracle не всегда автоматически устанавливает **DBMS_PROFILER** в вашей системе. Чтобы проверить доступность пакета **DBMS_PROFILER**, подключитесь к своей схеме в **SQL*Plus** и введите следующую команду:

```
DESC DBMS_PROFILER
```

Если вы увидите сообщение:

```
ERROR:
ORA-04043: object dbms_profiler does not exist
```

значит, вам (или вашему администратору базы данных) придется установить программу. Для этого запустите файл **\$ORACLE_HOME/rdbms/admin/profload.sql** с правами **SYSDBA**.

Затем выполните файл **\$ORACLE_HOME/rdbms/admin/proftab.sql** в вашей схеме, чтобы создать три таблицы, заполняемые данными **DBMS_PROFILER**:

- **PLSQL_PROFILER_RUNS** — родительская таблица запусков.
- **PLSQL_PROFILER_UNITS** — выполняемые программные модули.
- **PLSQL_PROFILER_DATA** — профилировка данных в каждой строке программного модуля.

После того как эти объекты будут определены, можно переходить к сбору профильной информации приложения, для чего пишется код следующего вида:

```
BEGIN
  DBMS_PROFILER.start_profiler (
    'my application' || TO_CHAR (SYSDATE, 'YYYYMMDD HH24:MI:SS')
  );

  код_приложения;
  DBMS_PROFILER.stop_profiler;
END;
```

После того как код приложения будет выполнен, вы можете обращаться с запросами к данным таблиц **PLSQL_PROFILER_**. Пример запроса, который выводит строки кода, на выполнение которых было затрачено не менее 1% общего времени выполнения:

```

/* Файл в Сети: slowest.sql */
SELECT      TO_CHAR (p1.total_time / 10000000, '99999999')
           || '-'
           || TO_CHAR (p1.total_occur)
           AS time_count,
           SUBSTR (p2.unit_owner, 1, 20)
           || '.'
           || DECODE (p2.unit_name,
                     '', '<anonymous>',
                     SUBSTR (p2.unit_name, 1, 20))
           AS unit,
           TO_CHAR (p1.line#) || '-' || p3.text text
FROM        plsql_profiler_data p1,
           plsql_profiler_units p2,
           all_source p3,
           (SELECT SUM (total_time) AS grand_total
            FROM plsql_profiler_units) p4
WHERE       p2.unit_owner NOT IN ('SYS', 'SYSTEM')
           AND p1.runid = &&firstparm
           AND (p1.total_time >= p4.grand_total / 100)
           AND p1.runid = p2.runid
           AND p2.unit_number = p1.unit_number
           AND p3.TYPE = 'PACKAGE BODY'
           AND p3.owner = p2.unit_owner
           AND p3.line = p1.line#
           AND p3.name = p2.unit_name
ORDER BY   p1.total_time DESC

```

Как видите, запросы получаются достаточно сложными (я изменил один из готовых запросов Oracle для получения четырехстороннего объединения). По этой причине гораздо лучше воспользоваться графическим интерфейсом программы разработки PL/SQL.

Иерархический профайлер

В Oracle Database 11g появился второй механизм профилирования: DBMS_HPROF, также называемый *иерархическим профайлером*. Используйте его для получения профиля выполнения кода PL/SQL, упорядоченного по вызовам подпрограмм вашего приложения. «Погодите, — скажете вы, — разве DBMS_PROFILER не делает это за меня?» Не совсем. Не-иерархические профайлеры (такие, как DBMS_PROFILER) сохраняют время, проведенное вашим приложением в каждой подпрограмме, вплоть до времени выполнения отдельных строк кода. Эта информация безусловно полезна, но у ее полезности есть свои ограничения. Часто требуется узнать, сколько времени приложение проводит в конкретной подпрограмме, — то есть «свернуть» профильную информацию до уровня подпрограммы. Именно это и делает новый иерархический профайлер.

Иерархический профайлер PL/SQL выдает информацию о затратах времени на выполнение каждой профилируемой подпрограммы приложения, различая время выполнения SQL и PL/SQL. Профайлер отслеживает разнообразную информацию, включая количество вызовов подпрограммы; проводимое в ней время; время, затраченное на поддерево подпрограммы (то есть с включением подпрограмм нижних уровней) и подробную информацию о связях «родитель — потомок».

Иерархический профайлер состоит из двух компонентов:

- *Сборщик данных* — предоставляет API для включения и выключения иерархического профилирования. Исполнительное ядро PL/SQL записывает низкоуровневый вывод профайлера в заданный файл.
- *Анализатор* — обрабатывает низкоуровневый вывод профайлера и сохраняет результаты в иерархических таблицах, к которым затем можно обращаться с запросами для вывода профильной информации.

Работа с иерархическим профайлером проходит примерно так:

1. Убедитесь в том, что вы можете выполнить пакет `DBMS_HPROF`.
2. Убедитесь в том, что у вас имеются привилегии `WRITE` для каталога, заданного при вызове `DBMS_HPROF.START_PROFILING`.
3. Создайте три таблицы профайлера (подробности см. ниже).
4. Вызовите процедуру `DBMS_HPROF.START_PROFILING`, чтобы запустить сбор данных иерархического профайлера в вашем сеансе.
5. Выполняйте код своего приложения достаточно долго и многократно, чтобы собранных данных было достаточно для получения интересных результатов.
6. Вызовите процедуру `DBMS_HPROF.STOP_PROFILING` для завершения сбора профильных данных.
7. Проанализируйте собранные данные и обратитесь с запросами к таблицам профайлера для получения результатов.



Чтобы получить самые точные данные о времени, потраченном при выполнении подпрограмм, сведите к минимуму всю постороннюю деятельность в системе, в которой выполняется приложение.

Конечно, в рабочей системе другие процессы могут замедлить работу вашей программы. Также желательно проводить сбор данных при использовании механизма RAT (Real Application Testing) в Oracle Database 11g и выше для получения реального времени отклика.

Чтобы создать таблицы профайлера и другие необходимые объекты базы данных, запустите сценарий `dbmshtab.sql` (находящийся в каталоге `$ORACLE_HOME/rdbms/admin`). Сценарий создает три таблицы:

- `DBMSHP_RUNS` — информация верхнего уровня о каждом запуске программы `ANALYZE` пакета `DBMS_HPROF`.
- `DBMSHP_FUNCTION_INFO` — подробная информация о выполнении каждой подпрограммы, профилированной при конкретном запуске программы `ANALYZE`.
- `DBMSHP_PARENT_CHILD_INFO` — информация «родитель — потомок» для каждой подпрограммы, профилированной в `DBMSHP_FUNCTION_INFO`.

Рассмотрим очень простой пример: допустим, я хочу протестировать производительность процедуры `intab` (предназначенной для вывода содержимого заданной таблицы средствами `DBMS_SQL`). Итак, сначала я включаю профилирование, указывая, что низкоуровневые данные должны записываться в файл `tab_trace.txt` в каталоге `TEMP_DIR`. Этот каталог должен быть предварительно определен командой `CREATE DIRECTORY`:

```
EXEC DBMS_HPROF.start_profiling ('TEMP_DIR', 'intab_trace.txt')
```

Затем вызывается программа (выполняется код приложения):

```
EXEC intab ('DEPARTMENTS')
```

Наконец, сеанс профилирования завершается:

```
EXEC DBMS_HPROF.stop_profiling;
```

Я мог бы включить все три команды в один блок кода; вместо этого я привел их раздельно, потому что обычно команды профилирования не должны размещаться в коде приложения или рядом с ним.

Итак, файл трассировки заполнен данными. Я могу открыть его и просмотреть данные — скорее всего, без особой пользы. Гораздо более эффективным применением моего времени и технологий Oracle будет вызов программы `ANALYZE` из пакета `DBMS_HPROF`.

Функция получает содержимое файла трассировки, преобразует данные и размещает их в трех таблицах. Она возвращает номер запуска, который затем должен использоваться для запросов к содержимому этих таблиц. Вызов `ANALYZE` выглядит следующим образом:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE (
    DBMS_HPROF.ANALYZE ('TEMP_DIR', 'intab_trace.txt'));
END;
/
```

Вот и все! Данные собраны, проанализированы и распределены по таблицам, и теперь я могу выбрать один из двух способов получения профильной информации:

1. Запустите программу командной строки `plshprof` (из каталога `$ORACLE_HOME/bin/`). Программа генерирует очень простые отчеты HTML по данным одного или двух файлов, содержащих низкоуровневый вывод профайлера. Пример низкоуровневого вывода профайлера приведен в разделе «Collecting Profile Data» руководства *Oracle Database Development Guide*. Далее сгенерированные отчеты в формате HTML просматриваются в любом браузере.
2. Выполните специально написанный запрос. Допустим, предыдущий блок возвращает номер запуска 177. Следующий запрос возвращает все текущие запуски:

```
SELECT runid, run_timestamp, total_elapsed_time, run_comment
FROM dbmshp_runs
```

Запрос для получения имен всех профилированных подпрограмм по всем запускам:

```
SELECT symbolid, owner, module, type, function, line#, namespace
FROM dbmshp_function_info
```

Запрос для вывода информации о выполнении подпрограмм для конкретного запуска:

```
SELECT FUNCTION, line#, namespace, subtree_elapsed_time
, function_elapsed_time, calls
FROM dbmshp_function_info
WHERE runid = 177
```

Этот запрос получает информацию «родитель — потомок» для текущего запуска, но результат не представляет особого интереса, так как пользователь видит только значения ключей, а не имена программ:

```
SELECT parentsymid, childsymid, subtree_elapsed_time, function_elapsed_time
, calls
FROM dbmshp_parent_child_info
WHERE runid = 177
```

Следующий запрос, полученный объединением с таблицей информации о функциях, уже более интересен; теперь пользователь видит имена родительских и дочерних программ, а также время их выполнения и количество вызовов:

```
SELECT      RPAD (' ', LEVEL * 2, ' ') || fi.owner || '.' || fi.module AS NAME
, fi.FUNCTION, pci.subtree_elapsed_time, pci.function_elapsed_time
, pci.calls
FROM dbmshp_parent_child_info pci JOIN dbmshp_function_info fi
ON pci.runid = fi.runid AND pci.childsymid = fi.symbolid
WHERE pci.runid = 177
CONNECT BY PRIOR childsymid = parentsymid
START WITH pci.parentsymid = 1
```

Иерархический профайлер — чрезвычайно мощная и полезная программа. Я рекомендую ознакомиться с ее подробным описанием в главе 13 *Oracle Database Development Guide*.

Хронометраж

Допустим, вы нашли «узкое место» своего приложения — какую-нибудь функцию со сложным алгоритмом, которая очевидным образом нуждается в оптимизации. Вы немного поработали над ней и теперь хотите знать, стала ли она быстрее работать. Конечно, для этого *можно* провести профилирование всего приложения, но намного удобнее было бы запустить исходную и измененную версии параллельно и посмотреть, какая работает быстрее. Для этого понадобится средство, вычисляющее время выполнения *отдельных программ* (и даже отдельных строк кода в программах).

Пакет DBMS_UTILITY содержит две функции для получения такой информации: DBMS_UTILITY.GET_TIME и DBMS_UTILITY.GET_CPU_TIME. Обе функции доступны в Oracle 10g и выше. С их помощью можно легко вычислить время, затраченное на выполнение блока кода (как общее, так и процессорное) с точностью до секунды. Общая схема выглядит так:

1. Вызовите DBMS_UTILITY.GET_TIME (или GET_CPU_TIME) перед выполнением кода.
2. Выполните код, для которого вы хотите получить хронометражные данные.
3. Вызовите DBMS_UTILITY.GET_TIME (или GET_CPU_TIME) для получения конечной точки интервала. Вычтите начальное время из конечного; разность определяет промежуток времени между этими двумя моментами (в сотых долях секунды).

Пример:

```
DECLARE
  l_start_time PLS_INTEGER;
BEGIN
  l_start_time := DBMS_UTILITY.get_time;

  my_program;

  DBMS_OUTPUT.put_line (
    'Elapsed: ' || DBMS_UTILITY.get_time - l_start_time);
END;
```

А теперь немного неожиданное замечание: эти функции чрезвычайно полезны, но их почти никогда не следует вызывать напрямую в сценариях. Вызовы (а также вычисление разности) желательно *инкапсулировать* в пакете или объектном типе. Хронометраж в этом случае выглядит примерно так:

```
BEGIN
  sf_timer.start_timer ();

  my_program;

  sf_timer.show_elapsed_time ('Программа my_program выполнена');
END;
```

Я рекомендую избегать прямых вызовов DBMS_UTILITY.GET_TIME и вместо них использовать пакет таймеров sf_timer по двум причинам:

- Повышение производительности: кому захочется вручную объявлять локальные переменные, писать код вызова нескольких встроенных функций и вычисления разности? Гораздо удобнее воспользоваться готовым решением.
- Обеспечение корректности данных: простая формула «конец — начало» иногда дает *отрицательный* результат!

Откуда может появиться отрицательное время? Число, возвращаемое DBMS_UTILITY.GET_TIME, представляет общее количество секунд, прошедших с некоторого момента времени. Если значение становится очень большим (конкретное значение зависит от операционной системы), оно обнуляется, и отсчет начинается заново. Если функция

GET_TIME будет вызвана непосредственно перед сбросом, разность окажется отрицательной!

Чтобы предотвратить отрицательные значения, приходится писать код следующего вида:

```
DECLARE
  c_big_number NUMBER := POWER (2, 32);
  l_start_time PLS_INTEGER;
BEGIN
  l_start_time := DBMS_UTILITY.get_time;
  my_program;
  DBMS_OUTPUT.put_line (
    'Elapsed: '
    || TO_CHAR (MOD (DBMS_UTILITY.get_time - l_start_time + c_big_number
                    , c_big_number)));
END;
```

Какой нормальный человек, которого к тому же постоянно подгоняет начальство, захочет писать такой код при каждом выполнении хронометража? По этой причине я создал пакет `sf_timer`, который скрывает технические подробности и упрощает анализ/сравнение результатов.

Выбор самой быстрой программы

Казалось бы, выбор самой быстрой программы должен быть четким и однозначным. Вы запускаете сценарий, смотрите, какая из реализаций работает быстрее других, и останавливаетесь на ней. Все хорошо, но в каких условиях вы запускали эти реализации? Даже если реализация С обеспечивает максимальную скорость в одном наборе условий, это не означает, что она будет всегда (или хотя бы достаточно часто) работать быстрее других реализаций.

При тестировании производительности, и особенно при выборе одной из нескольких реализаций, необходимо рассмотреть и протестировать следующие сценарии:

- **Положительные результаты** — программа получает действительные входные данные и делает то, что ей положено делать.
- **Отрицательные результаты** — программа получает недействительные входные данные (скажем, несуществующий первичный ключ) и не может выполнить необходимых действий.
- **Нейтральность алгоритма по отношению к данным** — программа хорошо работает с таблицей из 10 строк... а если таблица содержит 10 000 строк? Программа ищет данные в коллекции, но как повлияет на производительность местонахождение совпадения — в начале, в середине или в конце коллекции?
- **Многопользовательское выполнение** — программа хорошо работает для одного пользователя, но ее следует проверить для параллельного многопользовательского доступа. Вы ведь не хотите, чтобы взаимные блокировки обнаружились уже после запуска продукта в эксплуатацию, не так ли?
- **Тестирование во всех поддерживаемых версиях Oracle** — например, если приложение должно надежно работать в Oracle10g и Oracle11g, вы должны выполнить хронометражные сценарии в экземплярах каждой из этих версий.

Конечно, конкретная реализация сценариев зависит от тестируемой программы. Возможно, стоит создать процедуру, которая выполняет каждую из реализаций и вычисляет время, затраченное на их выполнение. Список параметров процедуры должен включать количество запусков каждой программы; получить полезные результаты от однократного запуска вряд ли удастся. Код необходимо выполнить достаточное количество раз, чтобы исходная загрузка кода и данных в память не искажала результаты. Другие параметры

процедуры определяются тем, какие данные необходимо передать программам для их выполнения.

Далее приведена заготовка такой процедуры с вызовами `sf_timer`:

```
/* Файл в Сети: compare_performance_template.sql */
PROCEDURE compare_implementations (
    title_in      IN VARCHAR2
    , iterations_in  IN INTEGER
)
/*
Параметры, необходимые для передачи данных сравниваемым программам...
*/
)
IS
BEGIN
    DBMS_OUTPUT.put_line ('Compare Performance of <CHANGE THIS>: ');
    DBMS_OUTPUT.put_line (title_in);
    DBMS_OUTPUT.put_line ('Each program execute ' || iterations_in || ' times.');
```

/*
Для каждой реализации запускается таймер, программа выполняется N раз,
после чего выводится затраченное время.
*/

```
    sf_timer.start_timer;

    FOR indx IN 1 .. iterations_in
    LOOP
        /* Вызов вашей программы. */
        NULL;
    END LOOP;

    sf_timer.show_elapsed_time ('<CHANGE THIS: Implementation 1');
    --
    sf_timer.start_timer;

    FOR indx IN 1 .. iterations_in
    LOOP
        /* Вызов вашей программы. */
        NULL;
    END LOOP;

    sf_timer.show_elapsed_time ('<CHANGE THIS: Implementation 2');
END compare_implementations;
```

Пакет `sf_timer` используется далее в некоторых примерах этой главы.

Предотвращение заикливания

Бесконечные циклы редко встречаются в окончательных версиях приложений (при условии, что ваша группа тестирования потрудились на совесть!) — эта проблема обычно встречается только в процессе разработки. Иногда приходится программировать хитрую логику завершения циклов, а аварийное прерывание сеанса в ходе тестирования отрицательно сказывается на эффективности.

Для борьбы с этим раздражающим недостатком был разработан пакет `Loop Killer`. Он базируется на простой идее: даже если вы не знаете, как успешно завершить цикл, вам может быть известно, что выполнение тела цикла более N раз (100, 1000 — в зависимости от ситуации) свидетельствует о возникшей проблеме.

Откомпилируйте пакет `Loop Killer` в схеме разработки и напишите небольшой фрагмент кода, который обеспечит прерывание цикла при достижении заданного порогового количества итераций.

Спецификация пакета выглядит так (полный код имеется на сайте книги):

```
/* Файл в Сети: sf_loop_killer.pks/pkb */
PACKAGE sf_loop_killer
IS
  c_max_iterations    CONSTANT PLS_INTEGER DEFAULT 1000;
  e_infinite_loop_detected    EXCEPTION;
  c_infinite_loop_detected    PLS_INTEGER := -20999;
  PRAGMA EXCEPTION_INIT (e_infinite_loop_detected, -20999);

  PROCEDURE kill_after (max_iterations_in IN PLS_INTEGER);

  PROCEDURE increment_or_kill (by_in IN PLS_INTEGER DEFAULT 1);

  FUNCTION current_count RETURN PLS_INTEGER;
END sf_loop_killer;
```

Простой пример использования: мы указываем, что цикл должен прерываться после 100 итераций, а затем вызываем `increment_or_kill` в конце тела цикла. При выполнении этого кода (в бесконечном цикле) генерируется необработанное исключение (рис. 21.1).

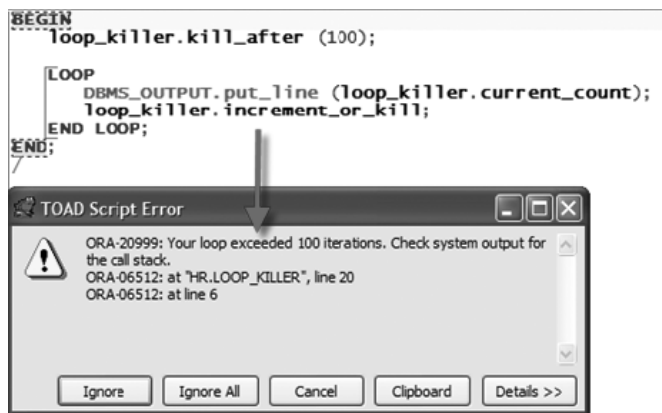


Рис. 21.1. Использование пакета Loop Killer

Предупреждения, относящиеся к производительности

Система предупреждений компилятора появилась в Oracle10g. При включении предупреждений в сеансе Oracle выдает информацию о качестве кода и советы по улучшению его удобочитаемости и производительности. Прислушайтесь к предупреждениям компилятора — они помогут вам выявить области кода, пригодные для оптимизации. Полный набор предупреждений, относящихся к производительности, включается следующей командой:

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:PERFORMANCE'
```

Дополнительная информация о предупреждениях и их использовании содержится в разделе «Предупреждения при компиляции» главы 20. Полный список предупреждений приведен в справочнике Error Messages документации Oracle.

Оптимизирующий компилятор

Оптимизирующий компилятор PL/SQL способен заметно ускорить выполнение кода за счет относительно невысоких затрат на стадии компиляции. Преимущества оптимизации распространяются как на интерпретируемый, так и компилируемый код PL/

SQL, поскольку оптимизации применяются на основании анализа закономерностей в исходном коде.

Оптимизирующий компилятор активен по умолчанию. Тем не менее иногда приходится изменять его поведение — снижать уровень оптимизации или даже полностью отключать ее. Например, если в ходе выполнения обычных операций системе приходится перекомпилировать большой объем кода или если приложение генерирует много строк динамически выполняемого кода PL/SQL, затраты ресурсов на оптимизацию могут оказаться неприемлемыми. Однако следует учитывать, что по тестам Oracle оптимизация в среднем вдвое повышает производительность кода PL/SQL, сопряженного с большим объемом вычислений.

В некоторых случаях оптимизация может даже изменить поведение программы. Примеры такого рода встречаются в коде, написанном для Oracle9i, зависящим от относительного порядка выполнения инициализации в нескольких пакетах. Если в ходе тестирования проявляются подобные проблемы, но при этом вы хотите пользоваться преимуществами оптимизации, вам придется переписать часть кода или определить блок инициализации, обеспечивающий нужный порядок выполнения.

Уровень оптимизации задается параметром `PLSQL_OPTIMIZE_LEVEL` (и соответствующими командами DDL `ALTER`), которому может быть присвоено значение 0, 1, 2 или 3 (последнее доступно только в Oracle11g). Чем выше уровень, тем больше усилий приложит компилятор.

Выберите уровень оптимизации, который лучше всего подходит для вашего приложения:

- `PLSQL_OPTIMIZE_LEVEL = 0` — оптимизация отключена. Компилятор PL/SQL сохраняет исходный порядок обработки команд, используемый в Oracle9i и более ранних версиях. Код будет работать быстрее, чем в старых версиях, но различия не столь значительны.
- `PLSQL_OPTIMIZE_LEVEL = 1` — компилятор применяет многочисленные оптимизации (такие, как удаление неиспользуемых вычислений и исключений). В общем случае порядок выполнения исходного кода остается неизменным.
- `PLSQL_OPTIMIZE_LEVEL = 2` — значение по умолчанию, самый высокий уровень оптимизации до Oracle11g. Применяет множество современных методов оптимизации, выходящих за рамки уровня 1, причем некоторые изменения могут привести к перемещению кода на достаточно большое расстояние от исходного местоположения. Оптимизация уровня 2 обеспечивает хороший выигрыш по быстродействию, но может значительно увеличить время компиляции некоторых программ.
- `PLSQL_OPTIMIZE_LEVEL = 3` — этот уровень оптимизации, появившийся только в Oracle11g, включает возможность подстановки (`inlining`) кода вложенных или локальных подпрограмм. Он может принести пользу в особых случаях (при большом количестве локальных подпрограмм или рекурсивных вызовов), но для большинства приложений PL/SQL должно хватить используемого по умолчанию уровня 2.

Уровень оптимизации можно задать для экземпляра в целом, а затем переопределить его для конкретного сеанса или программы. Пример:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 0;
```

Oracle поддерживает настройки оптимизатора на уровне отдельных модулей. При перекомпиляции модуля с настройками, отличными от настроек по умолчанию, новые значения «закрепляются», что позволяет позднее использовать их конструкцией `REUSE SETTINGS`:

```
ALTER PROCEDURE bigproc COMPILE PLSQL_OPTIMIZE_LEVEL = 0;
```

и затем:

```
ALTER PROCEDURE bigproc COMPILE REUSE SETTINGS;
```

Чтобы просмотреть все настройки компилятора для ваших модулей, включая уровень оптимизации, режим (интерпретация или код для машинной платформы) и уровни предупреждений компилятора, обратитесь с запросом к представлению `USER_PLSQL_OBJECT_SETTINGS`.

Как работает оптимизатор

Оптимизаторы не только могут делать то, что запрещено обычным программистам, — они также способны выявлять и использовать закономерности в вашем коде, которые могут остаться незамеченными для вас. Один из главных приемов, применяемых оптимизаторами, заключается в изменении порядка выполняемых операций для повышения эффективности исполнения. Определение языка программирования устанавливает границы для перестановок, которые могут выполняться оптимизатором, но определение PL/SQL оставляет немалую свободу действий для оптимизатора. В оставшейся части этого раздела обсуждаются некоторые возможности PL/SQL с приведением примеров кода, который может выиграть от их применения.

Для начала рассмотрим пример с инвариантом цикла — операцией, которая выполняется в цикле, но остается неизменной во всех итерациях. Любой хороший программист взглянет на следующий фрагмент:

```
FOR e IN (SELECT * FROM employees WHERE DEPT = p_dept)
LOOP
    DBMS_OUTPUT.PUT_LINE('<DEPT>' || p_dept || '</DEPT>');
    DBMS_OUTPUT.PUT_LINE('<emp ID="' || e.empno || '">');
    etc.
END LOOP;
```

и скажет вам, что код будет выполняться быстрее, если вывести «инвариант» из цикла для предотвращения его лишних выполнений:

```
l_dept_str := '<DEPT>' || p_dept || '</DEPT>'
FOR e IN (SELECT * FROM employees WHERE DEPT = p_dept)
LOOP
    DBMS_OUTPUT.PUT_LINE(l_dept_str);
    DBMS_OUTPUT.PUT_LINE('<emp ID="' || e.empno || '">');
    etc.
END LOOP;
```

Впрочем, даже хороший программист может решить, что ясность кода первой версии предпочтительнее эффективности второй. Начиная с Oracle Database 10g PL/SQL уже не заставляет вас выбирать между ними. Со стандартными настройками оптимизатора компилятор выявляет закономерность в первой версии и преобразует ее в байт-код, реализующий вторую версию. Это возможно благодаря тому, что определение языка не требует многократного выполнения инвариантов в циклах; это одна из возможностей, которую оптимизатор может использовать (и использует). Возможно, вам кажется, что эффект от такой оптимизации незначителен, но даже мелочи способны накапливаться. Я еще не видел ни одной базы данных, которая бы становилась меньше со временем. Многие программы PL/SQL перебирают все записи растущей таблицы, а таблица с миллионом строк уже не считается чем-то необычным. Лично я буду очень рад, если Oracle автоматически устранил миллион лишних операций из моего кода.

Другой пример — рассмотрим следующую серию команд:

```
result1 := r * s * t;
...
result2 := r * s * v;
```

Если значения `r` и `s` не могут изменяться между этими двумя командами, PL/SQL может откомпилировать код в следующем виде:

```

interim := r * s;
result1 := interim * t;
...
result2 := interim * v;

```

Оптимизатор поступит так, если он решит, что сохранение значения во временной переменной будет выполнено быстрее, чем повторное умножение.

Oracle делится этой и другой информацией об оптимизаторе PL/SQL в статье «Freedom, Order, and PL/SQL Compilation», доступной в Oracle Technology Network (введите название статьи в поле поиска). Основные положения статьи:

- Если ваш код не требует выполнения фрагмента в строго определенном порядке по правилам ускоренного вычисления выражений или следования команд, PL/SQL может выполнить фрагмент в порядке, отличном от порядка его написания. Проявления этого изменения порядка могут быть разными: в частности, оптимизатор может изменить порядок выполнения инициализационных разделов пакета, а если вызывающей программе требуется только доступ к константе из пакета — компилятор просто сохраняет эту константу в вызывающем коде.
- PL/SQL рассматривает вычисление индексов массивов и идентификацию полей записей как операторы. Если у вас имеется вложенная коллекция записей и вы обращаетесь к определенному элементу и полю (например, `price(product)(type).settle`), PL/SQL должен определить внутренний адрес, связанный с переменной. Этот адрес интерпретируется как выражение; он может быть сохранен и использован позднее в программе для предотвращения затрат, связанных с повторными вычислениями.
- Как было показано ранее, PL/SQL может вводить промежуточные значения для предотвращения вычислений.
- PL/SQL может полностью исключать некоторые операции (например, `x*0`). При этом явный вызов функции не исключается; в выражении `f()*0` функция `f()` всегда будет вызываться при наличии побочных эффектов.
- PL/SQL не вводит новые исключения.
- PL/SQL может снять необходимость в инициировании исключений. Например, исключение деления на 0 в следующем коде может быть удалено, потому что оно недостижимо:

```
IF FALSE THEN y := x/0; END IF;
```

PL/SQL не предоставляет возможности сменить обработчик для заданного исключения.

Пункт 1 заслуживает более подробно рассмотрения. В написанных мной приложениях я привык пользоваться разделами инициализации пакетов, но я никогда не беспокоился о порядке их выполнения. Мои инициализационные разделы обычно были небольшими, и в них выполнялось присваивание статических значений (обычно читаемых из базы данных); эти операции не зависели от порядка выполнения. Если ваше приложение должно гарантировать порядок выполнения, лучше вывести код из инициализационного раздела и поместить его в отдельные функции инициализации, вызываемые явно. Например, вызов может выглядеть так:

```

pkgA.init();
pkgB.init();

```

В этом случае сначала выполняется инициализация `pkgA`, а затем инициализация `pkgB`. Данная рекомендация остается истинной, даже если вы не используете оптимизирующий компилятор.

О пункте 2 с примером `price(product)(type).settle` тоже стоит упомянуть особо. Если эта ссылка используется многократно с разными значениями переменной `type`, но

одинаковыми значениями переменной `product`, оптимизация может разбить адресацию на две части — первая вычисляет `price(product)`, а вторая (используемая в нескольких разных местах) вычисляет оставшуюся часть. Такой код выполняется быстрее, потому что при каждом использовании ссылки вычисляется только переменная часть адреса. Еще важнее другое — такие изменения легко вносятся компилятором, но из-за семантики PL/SQL внести их программисту в исходном коде будет очень трудно. Большая часть изменений оптимизации относится к этой категории; компилятор «за кулисами» легко делает то, что будет трудно сделать программисту.

PL/SQL включает другие возможности для выявления и ускорения некоторых идиом программирования. В команде

```
counter := counter + 1;
```

компилятор не генерирует машинный код полного сложения. Вместо этого PL/SQL распознает идиому программирования и использует специальную команду виртуальной машины PL/SQL (PVM) — «инкремент». Эта команда выполняется намного быстрее традиционного сложения (это относится к подмножеству числовых типов данных — `PLS_INTEGER` и `SIMPLE_INTEGER`, но не к `NUMBER`).

Также существует специальная команда для кода, выполняющего конкатенацию нескольких операндов:

```
str := 'value1' || 'value2' || 'value3' ...
```

Вместо того чтобы рассматривать происходящее как серию попарных конкатенаций, компилятор и PVM совместно выполняют серию конкатенаций в одной команде.

Большая часть замен, выполняемых оптимизатором, остается незамеченной для вас. Во время обновления может обнаружиться программа, которая работает хуже, чем вы рассчитывали, — потому что она полагается на порядок выполнения, который был изменен компилятором. Скорее всего, проблемы следует искать в порядке инициализации пакетов, но конечно, у вас все может быть по-другому.

И последнее замечание: способ модификации кода оптимизатором детерминирован, по крайней мере для заданного значения `PLSQL_OPTIMIZE_LEVEL`. Другими словами, если вы пишете, компилируете и тестируете свою программу — скажем, с принятым по умолчанию уровнем оптимизации 2, ее поведение не изменится с переходом на другой компьютер или другую базу данных — при условии, что версия базы данных и уровень оптимизации остались неизменными.

Оптимизация циклов выборки данных

Для версий до Oracle9i Database Release 2 включительно цикл `FOR` с курсором наподобие приведенного ниже будет получать ровно одну логическую строку за операцию выборки:

```
FOR arow IN (SELECT something FROM somewhere)
LOOP
  ...
END LOOP;
```

Итак, если команда должна получить 500 строк, операция выборки будет выполнена 500 раз, что потребует 500 высокочастотных «переключений контекста» между PL/SQL и SQL.

Однако начиная с Oracle Database 10g база данных выполняет автоматическую «массовую выборку», так что каждая выборка получает (до) 100 строк. Предыдущий цикл `FOR` с курсором использует только 5 операций выборки для получения 500 строк от ядра SQL. Все выглядит так, словно база данных автоматически перепрограммирует цикл с использованием конструкции `BULK COLLECT` (см. далее в этой главе).

Эта явно недокументированная функция также работает с кодом в следующей форме:

```
FOR arow IN cursorname  
LOOP  
    ...  
END LOOP;
```

Однако она не будет работать для кода в следующей форме:

```
OPEN cursorname;  
LOOP  
    EXIT WHEN cursorname%NOTFOUND;  
    FETCH cursorname INTO ...  
END LOOP;  
CLOSE cursorname;
```

Тем не менее эта внутренняя оптимизация обеспечивает большой выигрыш для циклов FOR с курсором (дополнительным преимуществом которых является компактность).

Кэширование данных

Чрезвычайно распространенный метод повышения производительности заключается в промежуточном кэшировании для данных, к которым программа часто обращается и которые могут (по крайней мере какое-то время) оставаться *статическими*, то есть неизменными.

Область SGA базы данных Oracle является «начальником всех кэшей». Это (обычно) очень большая и (всегда) очень сложная область памяти, которая служит посредником между базой данных (файлами на диске) и программами, работающими с этой базой данных.

Как упоминалось в главе 20, в SGA кэшируется следующая информация (а также много другой информации, но эта наиболее актуальна для программистов PL/SQL):

- Разобранные курсоры.
- Данные, запрашиваемые курсорами из базы данных.
- Частично откомпилированные представления программ.

В общем и целом база данных не использует SGA для кэширования *данных программ*. Когда вы объявляете переменную в своей программе, память для этих данных выделяется из PGA (для выделенного сервера). Каждое подключение к базе данных имеет собственную область PGA; таким образом, память, необходимая для хранения данных программы, копируется в каждом подключении, в котором эта программа вызывается.

К счастью, у использования памяти PGA имеется преимущество: программа PL/SQL может получать информацию из PGA намного быстрее, чем из SGA. Следовательно, кэширование в PGA открывает интересные возможности для повышения производительности. Oracle также предоставляет другие механизмы кэширования, специфические для PL/SQL и предназначенные для повышения производительности ваших программ. В этом разделе будут рассмотрены три типа кэширования PL/SQL (еще один механизм, на который также стоит обратить внимание, использует *контексты приложений*):

- **Пакетное кэширование** — область памяти *UGA*, используемая для хранения статических данных, выборка которых будет осуществляться многократно. Используйте программы PL/SQL для предотвращения повторного доступа к данным через уровень SQL в SGA. Это самый быстрый способ кэширования, но он же устанавливает самые жесткие ограничения в отношении обстоятельств, в которых он может безопасно использоваться.
- **Кэширование детерминированных функций** — если функция, объявленная как детерминированная, вызывается в команде SQL, Oracle кэширует входные данные

функции и ее возвращаемое значение. Если вызвать функцию с теми же входными данными, Oracle вернет сохраненное значение без вызова функции.

- **Кэширование результатов функций (Oracle Database 11g и выше)** — последнее достижение в области кэширования PL/SQL, являющееся одновременно самым интересным и самым полезным. Включая простую декларативную секцию в заголовке функции, вы можете приказать базе данных кэшировать входные данные и возвращаемые значения функции. Однако в отличие от детерминированного подхода, кэш результатов функций используется при любых вызовах функций (не только из команд SQL), а автоматически становится недействительным при изменении зависимых данных.



При использовании пакетного кэширования в системе сохраняется копия данных. Вы должны быть твердо уверены в том, что эта копия актуальна. Любой из методов кэширования может использоваться неправильно, в результате чего пользователи начнут получать некорректные данные.

Пакетное кэширование

Пакетный кэш состоит из одной или нескольких переменных, объявленных на уровне пакета (а не в подпрограмме этого пакета). Пакетное кэширование является одним из первых кандидатов на организацию кэширования, потому что такие данные сохраняются в сеансе, даже если программы этого сеанса в настоящий момент не используют данные и не вызывают никакие подпрограммы пакета. Иначе говоря, если вы объявляете переменную на уровне пакета, то присвоенное этой переменной значение будет сохраняться до отключения, перекомпиляции пакета или изменения значения.

Наше знакомство с пакетным кэшированием начнется с описания типичных ситуаций, в которых уместно применение этого способа кэширования. Затем будет рассмотрен простой пример с кэшированием одного значения. В завершение я покажу, как кэшировать в пакете реляционную таблицу (частично или полностью), с существенным ускорением доступа к данным этой таблицы.

Когда используется пакетное кэширование

Пакетное кэширование уместно использовать при выполнении следующих условий:

- Вы еще не перешли на Oracle Database 11g и выше. При разработке приложений для последних версий почти всегда стоит применять кэширование результатов функций вместо пакетного кэширования.
- Кэшируемые данные не изменяются на протяжении периода времени, в котором эти данные необходимы пользователю. Примеры статических данных — небольшие справочные таблицы, которые изменяются редко (а то и вовсе не изменяются), и пакетные сценарии, создающие «снимок» данных на момент запуска сценария; этот «снимок» используется до завершения сценария.
- Ваш сервер базы данных располагает достаточной памятью для хранения копии кэша для каждого сеанса подключения (и его использования). Для оценки размера кэша, определенного в вашем пакете, можно воспользоваться средствами, описанными ранее в этой главе.

И наоборот, пакетное кэширование *не следует* применять при выполнении хотя бы одного из следующих условий:

- Кэшируемые данные могут измениться за время работы пользователя с кэшем.
- Объем кэшируемых данных приводит к слишком высоким затратам памяти на сеанс, в результате чего при большом количестве пользователей происходят ошибки памяти.

Простой пример пакетного кэширования

Как известно, функция `USER` возвращает имя текущего сеанса. Oracle реализует эту функцию в пакете `STANDARD` следующим образом:

```
function USER return varchar2 is
c varchar2(255);
begin
    select user into c from sys.dual;
    return c;
end;
```

Таким образом, при каждом вызове `USER` выполняется запрос. Конечно, он выполняется быстро, но такой запрос никогда не должен выполняться более одного раза за сеанс, потому что возвращаемое значение остается неизменным. Возможно, вы спросите: ну и что? Не только запрос `SELECT FROM dual` выполняется очень эффективно, но и база данных Oracle также кэширует разобранный запрос и возвращенное значение, так что происходящее уже оптимизировано. Разве пакетное кэширование на что-нибудь повлияет? Еще как!

Возьмем следующий пакет:

```
/* Файл в Сети: thisuser.pkg */
PACKAGE thisuser
IS
    cname CONSTANT VARCHAR2(30) := USER;
    FUNCTION name RETURN VARCHAR2;
END;

PACKAGE BODY thisuser
IS
    g_user VARCHAR2(30) := USER;

    FUNCTION name RETURN VARCHAR2 IS BEGIN RETURN g_user; END;
END;
```

Я кэширую значение, возвращаемое `USER`, двумя разными способами:

- **Константа, определяемая на уровне пакета.** Исполнительное ядро PL/SQL вызывает `USER` для инициализации константы при инициализации пакета (при первом использовании).
- **Функция.** Функция возвращает имя «текущего пользователя» — значение, возвращаемое функцией, берется из приватной переменной (в теле пакета); оно присваивается при инициализации пакета по результату вызова `USER`.

Итак, кэши созданы, и теперь мы можем проверить, стоит ли игра свеч. Будет ли какая-либо из реализаций работать значительно быстрее по сравнению с многократным вызовом высокооптимизированной функции `USER`?

Следующий сценарий использует пакет `sf_timer` для сравнения производительности:

```
/* Файл в Сети: thisuser.tst */
PROCEDURE test_thisuser (count_in IN PLS_INTEGER)
IS
    l_name all_users.username%TYPE;
BEGIN
    sf_timer.start_timer;
    FOR indx IN 1 .. count_in LOOP l_name := thisuser.NAME; END LOOP;
    sf_timer.show_elapsed_time ('Packaged Function');
    --
    sf_timer.start_timer;
    FOR indx IN 1 .. count_in LOOP l_name := thisuser.cname; END LOOP;
    sf_timer.show_elapsed_time ('Packaged Constant');
    --
```

```

sf_timer.start_timer;
FOR indx IN 1 .. count_in LOOP l_name := USER; END LOOP;
sf_timer.show_elapsed_time ('USER Function');
END test_thisuser;

```

Если запустить его сначала для 100, а затем для 1 000 000 итераций, будут получены следующие результаты:

```

Packaged Function Elapsed: 0 seconds.
Packaged Constant Elapsed: 0 seconds.
USER Function Elapsed: 0 seconds.

```

```

Packaged Function Elapsed: .48 seconds.
Packaged Constant Elapsed: .06 seconds.
USER Function Elapsed: 32.6 seconds.

```

Результат очевиден: при малом количестве итераций преимущества кэширования не очевидны. Однако с увеличением количества итераций пакетное кэширование работает значительно быстрее, чем вызов через уровень SQL и SGA.

Кроме того, обращение к константе происходит значительно быстрее, чем вызов функции, возвращающий значение. Тогда зачем использовать функцию? Версия с функцией обладает важным преимуществом перед константой: она скрывает значение. Таким образом, если по какой-либо причине значение потребуется изменить (неприменимо к нашему сценарию), это можно сделать без перекомпиляции спецификации пакета, которая бы потребовала перекомпиляции всех программ, зависящих от пакета.

И хотя вам вряд ли когда-нибудь потребуется кэшировать значение, возвращаемое функцией USER, надеюсь, вы убедились в том, что пакетное кэширование является весьма эффективным способом хранения и выборки данных. А теперь рассмотрим менее тривиальный пример.

Кэширование содержимого таблицы в пакете

Если приложение содержит таблицу, которая никогда не изменяется в нормальное рабочее время (то есть остается статической во время работы с ней), вы можете легко создать пакет, кэширующий все содержимое таблицы. Скорость обработки запросов при этом вырастет на порядок и более.

Допустим, имеется статическая таблица `products`, которая определяется следующим образом:

```

/* Файл в Сети: package_cache_demo.sql */
TABLE products (
  product_number INTEGER PRIMARY KEY
, description VARCHAR2(1000))

```

Ниже приведено тело пакета, предоставляющего два механизма получения данных из таблицы — с выполнением запроса и кэшированием данных с последующей выборкой из кэша:

```

1 PACKAGE BODY products_cache
2 IS
3   TYPE cache_t IS TABLE OF products%ROWTYPE INDEX BY PLS_INTEGER;
4   g_cache cache_t;
5
6   FUNCTION with_sql (product_number_in IN products.product_number%TYPE)
7     RETURN products%ROWTYPE
8   IS
9     l_row products%ROWTYPE;
10  BEGIN
11    SELECT * INTO l_row FROM products

```

продолжение ➤

```

12         WHERE product_number = product_number_in;
13         RETURN l_row;
14     END with_sql;
15
16     FUNCTION from_cache (product_number_in IN products.product_number%TYPE)
17         RETURN products%ROWTYPE
18     IS
19     BEGIN
20         RETURN g_cache (product_number_in);
21     END from_cache;
22 BEGIN
23     FOR product_rec IN (SELECT * FROM products) LOOP
24         g_cache (product_rec.product_number) := product_rec;
25     END LOOP;
26 END products_cache;

```

В следующей таблице описаны основные аспекты этого пакета.

| Строки | Описание |
|--------|--|
| 3–4 | Объявление ассоциативного массива g_cache, который воспроизводит структуру таблицы products; каждый элемент коллекции представляет собой запись с такой же структурой, как у строки таблицы |
| 6–14 | Функция with_sql возвращает одну строку из таблицы products для заданного первичного ключа с использованием «традиционного» метода SELECT INTO. Иначе говоря, при каждом вызове этой функции выполняется запрос |
| 16–21 | Функция from_cache также возвращает одну строку из таблицы products для заданного первичного ключа, но при этом значение первичного ключа используется как индекс, а строка извлекается из g_cache |
| 23–25 | При инициализации пакета содержимое таблицы products загружается в коллекцию g_cache. Обратите внимание: значение первичного ключа используется как индекс в коллекции. Именно такой способ работы с первичным ключом делает возможной (и такой удобной) реализацию from_cache |

Когда пользователь впервые вызовет функцию from_cache (или with_sql), база данных сначала выполнит этот код.

Затем я конструирую и выполняю блок кода для сравнения производительности двух решений:

```

DECLARE
    l_row    products%ROWTYPE;
BEGIN
    sf_timer.start_timer;
    FOR indx IN 1 .. 100000
    LOOP
        l_row := products_cache.from_cache (5000);
    END LOOP;
    sf_timer.show_elapsed_time ('Cache table');
    --
    sf_timer.start_timer;
    FOR indx IN 1 .. 100000
    LOOP
        l_row := products_cache.with_sql (5000);
    END LOOP;
    sf_timer.show_elapsed_time ('Run query every time');
END;

```

Результаты выполнения:

Cache table Elapsed: .14 seconds.

Run query every time Elapsed: 4.7 seconds.

И снова предельно очевидно, что пакетное кэширование работает намного, намного быстрее, чем многократное повторение запроса — даже если запрос полностью оптимизирован со всей мощностью и сложностью SGA.

Избирательное кэширование табличных данных

Допустим, я нашел статическую таблицу, к которой хотелось бы применить этот способ кэширования. Однако возникает одна проблема: таблица содержит 100 000 строк данных. Можно создать пакет наподобие `products_cache` из предыдущего раздела, но он будет использовать 5 Мбайт памяти в PGA каждого сеанса. С 500 параллельными подключениями кэш займет 2,5 Гбайт, а это неприемлемо. К счастью, я замечаю, что каждый пользователь обычно работает примерно с 50 строками данных. Следовательно, кэширование всей таблицы в каждом сеансе расточительно в отношении как вычислительных ресурсов (исходная загрузка 100 000 записей), так и памяти.

Если содержимое таблицы статично, но вам не нужны *все* данные из этой таблицы, рассмотрите возможность применения *избирательного* кэширования. Это означает, что при инициализации пакета содержимое таблицы не загружается в кэш. Если запрашиваемая пользователем строка находится в кэше, она возвращается немедленно, а если нет — вы запрашиваете эту одну строку из таблицы, добавляете ее в кэш, а затем возвращаете данные.

Когда пользователь в следующий раз запросит ту же строку, она будет извлечена из кэша. Следующий фрагмент демонстрирует этот способ:

```
/* Файл в Сети: package_cache_demo.sql */
FUNCTION jit_from_cache (product_number_in IN products.product_number%TYPE)
    RETURN products%ROWTYPE
IS
    l_row    products%ROWTYPE;
BEGIN
    IF g_cache.EXISTS (product_number_in)
    THEN
        /* Строка уже находится в кэше, возвращаем ее. */
        l_row := g_cache (product_number_in);
    ELSE
        /* Первый запрос, извлекаем строку из базы данных
           и добавляем ее в кэш. */
        l_row := with_sql (product_number_in);
        g_cache (product_number_in) := l_row;
    END IF;
    RETURN l_row;
END jit_from_cache;
```

В общем случае избирательное кэширование работает медленнее одноразовой загрузки всех данных в кэш. Тем не менее этот способ все равно значительно превосходит по скорости повторное обращение к базе данных.

Кэширование детерминированных функций

Функция называется детерминированной, если она возвращает одинаковый результат при всех вызовах с одинаковыми значениями аргументов `IN` и `IN OUT`. Также можно рассматривать детерминированные программы с точки зрения отсутствия побочных эффектов: все изменения, вносимые программой, отражаются в списке параметров. За дополнительной информацией о детерминированных функциях обращайтесь к главе 17.

Именно из-за однозначности поведения детерминированных функций Oracle может построить кэш с входными данными и результатами функции. В конце концов, если для одного набора входных данных всегда выдается одинаковый результат, вызывать функцию повторно для того же набора параметров неразумно.

Рассмотрим еще один пример кэширования на базе детерминированных функций. Допустим, я определил следующую инкапсуляцию для функции `SUBSTR` (возвращает строку между заданной начальной и конечной позициями) как детерминированную функцию:

```

/* Файл в Сети: deterministic_demo.sql */
FUNCTION betwnstr (
  string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)
  RETURN VARCHAR2 DETERMINISTIC
IS
BEGIN
  RETURN (SUBSTR (string_in, start_in, end_in - start_in + 1));
END betwnstr;

```

Затем эта функция вызывается внутри запроса (она не изменяет никакие таблицы базы данных, в противном случае ее нельзя было бы использовать подобным образом):

```

SELECT betwnstr (last_name, 1, 5) first_five
FROM employees

```

При таком вызове **betwnstr** база данных создает кэш с входными данными и возвращаемыми значениями. Затем, если я снова вызову функцию с теми же данными, база данных сразу вернет значение без вызова функции. Чтобы продемонстрировать эффект этой оптимизации, я изменю реализацию **betwnstr**:

```

FUNCTION betwnstr (
  string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)
  RETURN VARCHAR2 DETERMINISTIC
IS
BEGIN
  DBMS_LOCK.sleep (.01);
  RETURN (SUBSTR (string_in, start_in, end_in - start_in + 1));
END betwnstr;

```

Подпрограмма **sleep** из пакета **DBMS_LOCK** приостанавливает **betwnstr** на сотую долю секунды.

Если вызвать эту функцию из блока кода PL/SQL (не из запроса), база данных не будет кэшировать возвращаемые значения, поэтому запрос 107 строк таблицы **employees** займет более секунды:

```

DECLARE
  l_string employees.last_name%TYPE;
BEGIN
  sf_timer.start_timer;
  FOR rec IN (SELECT * FROM employees)
  LOOP
    l_string := betwnstr ('FEUERSTEIN', 1, 5);
  END LOOP;

  sf_timer.show_elapsed_time ('Deterministic function in block');
END;
/

```

Результат:

Deterministic function in block Elapsed: 1.67 seconds.

Если теперь выполнить ту же логику, но переместить вызов **betwnstr** в запрос, ситуация существенно изменяется:

```

BEGIN
  sf_timer.start_timer;
  FOR rec IN (SELECT betwnstr ('FEUERSTEIN', 1, 5) FROM employees)
  LOOP
    NULL;
  END LOOP;
  sf_timer.show_elapsed_time ('Deterministic function in query');
END;
/

```


Результат:

Deterministic function in query Elapsed: .05 seconds.

Как видите, кэширование детерминированных функций является очень эффективным способом оптимизации. Главное — убедитесь в следующем:

- Если вы объявляете функцию детерминированной, убедитесь в том, что она действительно детерминирована. База данных Oracle не анализирует программный код, чтобы убедиться в том, что вы не ошиблись. Если добавить ключевое слово **DETERMINISTIC** в функцию, которая, например, запрашивает данные из таблицы, кэширование будет осуществляться некорректно, и пользователь получит искаженные данные.
- Для реализации эффекта детерминированного кэширования функция должна вызываться в команде **SQL**; это существенное ограничение полезности данного типа кэширования.

Кэширование результатов функций (Oracle Database 11g)

До выхода Oracle Database 11g пакетное кэширование было самым лучшим и гибким способом кэширования данных для использования в программах PL/SQL. К сожалению, обстоятельства, в которых оно может использоваться, ограничены — источник данных должен быть статическим, а затраты памяти растут с каждым новым сеансом, подключенным к базе данных Oracle.

Так как преимущества этого вида кэширования (а также кэширования, реализованного для детерминированных функций) были очевидны, компания Oracle реализовала кэширование результатов функций в Oracle Database 11g. Эта возможность предоставляет механизм кэширования, лишенный недостатков пакетного кэширования и почти не уступающий ему по скорости.

Включение кэширования результатов функций для конкретной функции дает следующие преимущества:

- Oracle сохраняет входные данные и соответствующие им возвращаемые значения в отдельном кэше для каждой функции. Кэш совместно используется всеми сеансами, подключенными к экземпляру базы данных; он не дублируется для каждого сеанса. В Oracle Database 11g Release 2 и выше кэш результатов функций даже совместно используется экземплярами в RAC (Real Application Cluster).
- При каждом вызове этой функции база данных проверяет, не был ли результат кэширован с теми же входными значениями. При наличии кэшированного результата функция не выполняется; вместо этого Oracle просто возвращает сохраненное значение.
- Каждый раз, когда изменения фиксируются в таблицах, идентифицированных как зависимости для кэшированных данных, база данных автоматически объявляет кэш недействительным. При последующих вызовах функции кэш заново заполняется согласованными данными.
- Кэширование происходит при каждом вызове функции; его не нужно специально активизировать для команд SQL.
- Нет необходимости писать код объявления и заполнения коллекций; вместо этого кэширование включается при помощи декларативного синтаксиса в заголовке функции.

Кэширование результатов чаще всего используется для функций, запрашивающих данные из таблиц. Хорошие кандидаты для кэширования результатов:

- Статические наборы данных (например, материализованные представления). Содержимое таких представлений не изменяется между обновлениями, так зачем производить выборку данных заново?
- Таблицы, частота запросов к которым существенно превышает частоту изменения. Если таблица изменяется в среднем каждые 5 минут, но между изменениями те же строки запрашиваются сотни и тысячи раз, кэш результатов может использоваться с пользой.

Но если таблица изменяется каждую секунду, кэширование результатов нежелательно; оно может только *замедлить приложение*, так как Oracle потратит слишком много времени на заполнение и последующую очистку кэша. Тщательно выбирайте, когда и как использовать эту возможность; поговорите с администратором базы данных и убедитесь в том, что размер пула SGA для кэша результатов достаточно велик для хранения всех данных, кэшируемых в режиме типичной нагрузки.

Далее я сначала опишу синтаксис кэширования результатов, а затем приведу простые примеры его использования. Также мы обсудим, в каких обстоятельствах следует использовать кэш результатов, упомянем аспекты управления кэшем, относящиеся к администрированию, и изучим основные ограничения и ловушки.

Включение кэширования результатов функций

Кэширование результатов функций включается очень просто — для этого в заголовок функции включается секция RESULT_CACHE. Обо всем остальном позаботится Oracle.

Синтаксис секции RESULT_CACHE:

```
RESULT_CACHE [ RELIES_ON (таблица_или_представление [, таблица_или_представление2 ...
таблица_или_представлениеN] ) ]
```

Секция RELIES_ON сообщает Oracle, от каких таблиц или представлений зависит содержимое кэша, и может включаться только в заголовки функций уровня схемы и в реализации пакетных функций (то есть в тело пакета). В Oracle Database 11g Release 2 она считается устаревшей. Ниже приведен пример пакетной функции — обратите внимание на то, что секция RESULT_CACHE должна присутствовать и в спецификации, и в теле:

```
CREATE OR REPLACE PACKAGE get_data
IS
    FUNCTION session_constant RETURN VARCHAR2 RESULT_CACHE;
END get_data;
/

CREATE OR REPLACE PACKAGE BODY get_data
IS
    FUNCTION session_constant RETURN VARCHAR2
        RESULT_CACHE
    IS
    BEGIN
        ...
    END session_constant;
END get_data;
/
```

Так просто и элегантно; вы добавляете всего одну секцию в заголовок своей функции, и приложение начинает работать заметно быстрее!

Секция RELIES_ON (считается устаревшей в 11.2)

Первое, что следует знать о RELIES_ON, — то, что в Oracle Database 11g Release 2 эта конструкция уже не нужна. Начиная с этой версии Oracle автоматически определяет, от каких таблиц зависят возвращаемые данные, и правильно объявляет недействительным

содержимое кэша при изменении содержимого этих таблиц; включение своей секции `RELIES_ON` ни к чему не приведет. Чтобы убедиться в этом, запустите сценарий `11gR2_frc_no_relies_on.sql` с сайта книги. В ходе анализа идентифицируются как таблицы, к которым программа обращается из статического (встроенного) и динамического SQL, так и таблицы, к которым она обращается только косвенно (через представления).

Но если вы работаете в Oracle Database 11g Release 1 или более ранней версии, вам придется явно перечислить все таблицы и представления, из которых берутся возвращаемые данные. Определить, какие таблицы и представления нужно включить в список, обычно бывает несложно. Если функция содержит команду `SELECT`, проследите за тем, чтобы все таблицы и представления из всех секций `FROM` запроса были включены в список.

При выборке из представления необходимо включить в список только само представление, а не таблицы, к которым оно обращается. Сценарий `11g_frc_views.sql`, также доступный на сайте книги, показывает, как база данных по определению представления находит все таблицы, при изменении которых содержимое кэша становится недействительным.

Несколько примеров использования `RELIES_ON`:

1. Функция уровня схемы с секцией `RELIES_ON` указывает, что кэш зависит от таблицы `employees`:

```
CREATE OR REPLACE FUNCTION name_for_id (id_in IN employees.employee_id%TYPE)
RETURN employees.last_name%TYPE
RESULT_CACHE RELIES ON (employees)
```

2. Пакетная функция с секцией `RELIES_ON` (может присутствовать только в теле):

```
CREATE OR REPLACE PACKAGE get_data
IS
    FUNCTION name_for_id (id_in IN employees.employee_id%TYPE)
        RETURN employees.last_name%TYPE
        RESULT_CACHE
END get_data;
/
```

```
CREATE OR REPLACE PACKAGE BODY get_data
IS
    FUNCTION name_for_id (id_in IN employees.employee_id%TYPE)
        RETURN employees.last_name%TYPE
        RESULT_CACHE RELIES ON (employees)
    IS
    BEGIN
        ...
    END name_for_id;
END get_data;
/
```

3. Секция `RELIES_ON` с перечислением нескольких объектов:

```
CREATE OR REPLACE PACKAGE BODY get_data
IS
    FUNCTION name_for_id (id_in IN employees.employee_id%TYPE)
        RETURN employees.last_name%TYPE
        RESULT_CACHE RELIES ON (employees, departments, locations)
    ...
```

Пример кэширования результатов: детерминированная функция

В предыдущем разделе вы узнали о кэшировании, связанном с детерминированными функциями. В частности, я заметил, что кэширование действует только при вызове функции внутри запроса. Теперь применим механизм кэширования результатов функций Oracle Database 11g к функции `betwnstr` и посмотрим, как он работает при вызове из блока PL/SQL.

В следующей функции в заголовок добавляется секция `RESULT_CACHE`. Я также добавил вызов `DBMS_OUTPUT.PUT_LINE`, который показывает, какие входные данные были переданы функции:

```
/* Файл в Сети: 11g_frc_simple_demo.sql */
FUNCTION betwnstr (
  string_in IN VARCHAR2, start_in IN INTEGER, end_in IN INTEGER)
  RETURN VARCHAR2 RESULT_CACHE
IS
BEGIN
  DBMS_OUTPUT.put_line (
    'betwnstr for ' || string_in || '-' || start_in || '-' || end_in);
  RETURN (SUBSTR (string_in, start_in, end_in - start_in + 1));
END;
```

Затем я вызываю эту функцию для 10 строк таблицы `employees`. Для четных идентификаторов работника к фамилии работника применяется функция `betwnstr`. В противном случае передается постоянный набор из трех значений:

```
DECLARE
  l_string employees.last_name%TYPE;
BEGIN
  FOR rec IN (SELECT * FROM employees WHERE ROWNUM < 11)
  LOOP
    l_string :=
      CASE MOD (rec.employee_id, 2)
        WHEN 0 THEN betwnstr (rec.last_name, 1, 5)
        ELSE betwnstr ('FEUERSTEIN', 1, 5)
      END;
    END LOOP;
  END;
```

При выполнении функции я получаю следующий результат:

```
betwnstr for OConnell-1-5
betwnstr for FEUERSTEIN-1-5
betwnstr for Whalen-1-5
betwnstr for Fay-1-5
betwnstr for Baer-1-5
betwnstr for Gietz-1-5
betwnstr for King-1-5
```

Обратите внимание: строка `FEUERSTEIN` встречается только один раз, хотя при вызове она использовалась пять раз. Этот пример демонстрирует практическое применение кэширования результатов.

Пример кэширования результатов функций: выборка данных из таблицы

Кэширование результатов функций чаще всего применяется при запросах данных из таблиц, содержимое которых чаще читается, чем изменяется (между изменениями данные остаются статическими). Предположим, в системе управления недвижимостью имеется таблица, в которой хранятся проценты для разных типов ссуд. Содержимое этой таблицы обновляется запланированным заданием, которое запускается ежедневно в течение дня. Структура таблицы и данные, которые будут использованы в демонстрационном сценарии:

```
/* Файл в Сети: 11g_frc_demo_table.sql */
CREATE TABLE loan_info (
  NAME VARCHAR2(100) PRIMARY KEY,
  length_of_loan INTEGER,
  initial_interest_rate NUMBER,
  regular_interest_rate NUMBER,
  percentage_down_payment INTEGER)
/
```

```

BEGIN
  INSERT INTO loan_info VALUES ('Five year fixed', 5, 6, 6, 20);
  INSERT INTO loan_info VALUES ('Ten year fixed', 10, 5.7, 5.7, 20);
  INSERT INTO loan_info VALUES ('Fifteen year fixed', 15, 5.5, 5.5, 10);
  INSERT INTO loan_info VALUES ('Thirty year fixed', 30, 5, 5, 10);
  INSERT INTO loan_info VALUES ('Two year balloon', 2, 3, 8, 0);
  INSERT INTO loan_info VALUES ('Five year balloon', 5, 4, 10, 5);
  COMMIT;
END;
/

```

Функция для получения всей информации в одной строке данных:

```

FUNCTION loan_info_for_name (NAME_IN IN VARCHAR2)
  RETURN loan_info%ROWTYPE
  RESULT_CACHE RELIES_ON (loan_info)
IS
  l_row loan_info%ROWTYPE;
BEGIN
  DBMS_OUTPUT.put_line ('> Looking up loan info for ' || NAME_IN);
  SELECT * INTO l_row FROM loan_info WHERE NAME = NAME_IN;
  RETURN l_row;
END loan_info_for_name;

```

В этом случае секция RESULT_CACHE включает внутреннюю секцию RELIES_ON, которой сообщает, что кэш для этой функции должен базироваться на («зависеть от») данных таблицы loan_info. Затем выполняется следующий сценарий, который вызывает функцию для двух разных имен, после чего изменяет содержимое таблицы, и наконец, снова вызывает функцию для одного из исходных имен:

```

DECLARE
  l_row loan_info%ROWTYPE;
BEGIN
  DBMS_OUTPUT.put_line ('First time for Five year fixed...');
  l_row := loan_info_for_name ('Five year fixed');
  DBMS_OUTPUT.put_line ('First time for Five year balloon...');
  l_row := loan_info_for_name ('Five year balloon');
  DBMS_OUTPUT.put_line ('Second time for Five year fixed...');
  l_row := loan_info_for_name ('Five year fixed');

  UPDATE loan_info SET percentage_down_payment = 25
    WHERE NAME = 'Thirty year fixed';
  COMMIT;
  DBMS_OUTPUT.put_line ('After commit, third time for Five year fixed...');
  l_row := loan_info_for_name ('Five year fixed');
END;

```

Результат, полученный при выполнении этого сценария:

```

First time for Five year fixed...
> Looking up loan info for Five year fixed
First time for Five year balloon...
> Looking up loan info for Five year balloon
Second time for Five year fixed...
After commit, third time for Five year fixed...
> Looking up loan info for Five year fixed

```

Несколько слов по поводу того, что же здесь происходит:

- При первом вызове функции для имени «Five year fixed» исполнительное ядро PL/SQL выполняет функцию, находит данные, помещает их в кэш и возвращает данные.
- При первом вызове функции для имени «Five year balloon» ядро выполняет функцию, находит данные, помещает их в кэш и возвращает данные.
- При втором вызове функции для имени «Five year fixed» функция не выполняется (во втором вызове нет сообщения «Looking up...»). Кэширование результата функции работает...

- Я изменяю значение столбца строки с именем «Thirty year fixed» и закрепляю изменение.
- Затем я *в третий раз* вызываю функцию для имени «Thirty year fixed». На этот раз функция снова выполняется с выдачей запроса к данным. Это происходит потому, что я сообщил Oracle, что содержимое кэша зависит от таблицы `loan_info`, а в эту таблицу были внесены изменения.

Пример кэширования результатов функций: кэширование коллекции

До настоящего момента я приводил примеры кэширования отдельных значений и целых записей. Кэшировать также можно целую коллекцию данных и даже коллекцию записей. В следующем коде я изменил функцию таким образом, чтобы она возвращала все имена ссуд в виде коллекции строк (на базе предопределенного типа коллекции `DBMS_SQL`). Затем я многократно вызываю эту функцию, но коллекция заполняется только один раз (конструкция `BULK COLLECT` рассматривается позднее в этой главе):

```
/* Файл в Сети: 11g_frc_table_demo.sql */
FUNCTION loan_names RETURN DBMS_SQL.VARCHAR2S
  RESULT_CACHE RELIES_ON (loan_info)
IS
  l_names  DBMS_SQL.VARCHAR2S;
BEGIN
  DBMS_OUTPUT.put_line ('> Looking up loan names....');
  SELECT name BULK COLLECT INTO l_names FROM loan_info;
  RETURN l_names;
END loan_names;
```

Следующий сценарий показывает, что механизм кэширования успешно работает даже при заполнении таких сложных типов:

```
DECLARE
  l_names  DBMS_SQL.VARCHAR2S;
BEGIN
  DBMS_OUTPUT.put_line ('First time retrieving all names...');
  l_names := loan_names ();
  DBMS_OUTPUT.put_line ('Second time retrieving all names...');
  l_names := loan_names ();

  UPDATE loan_info SET percentage_down_payment = 25
    WHERE NAME = 'Thirty year fixed';

  COMMIT;
  DBMS_OUTPUT.put_line ('After commit, third time retrieving all names...');
  l_names := loan_names ();
END;
```

Результат:

```
First time retrieving all names...
> Looking up loan names....
Second time retrieving all names...
After commit, third time retrieving all names...
> Looking up loan names....
```

Когда применяется кэширование результатов функций

К кэшированию всегда следует подходить с величайшей осторожностью. Если оно будет организовано неправильно, приложение может передать пользователю некорректные данные. Кэширование результатов функций — самый гибкий и часто применяемый из разных типов кэширования, встречающихся в коде PL/SQL, но и с ним возможны неприятности.

Рассмотрите возможность включения `RESULT_CACHE` в заголовок функции в следующих обстоятельствах:

- Данные запрашиваются из таблицы чаще, чем обновляются. Например, допустим, что в приложении для отдела кадров пользователи запрашивают содержимое таблицы `employees` тысячи раз в минуту, но обновление происходит в среднем каждые 10 минут. Между изменениями таблица `employees` остается статической, поэтому данные могут безопасно кэшироваться — с сокращением времени обработки запроса.
- Функция, которая не запрашивает данные, вызывается многократно (часто рекурсивно) с одинаковыми входными значениями. Классический пример из области программирования — алгоритм Фибоначчи. Чтобы вычислить число Фибоначчи для целого n (то есть $F(n)$), необходимо вычислить значения от $F(1)$ до $F(n-1)$.
- Ваше приложение (или каждый пользователь приложения) зависит от параметров конфигурации, которые остаются статическими во время работы с приложением: идеальная ситуация для кэширования результатов!

Когда кэширование результатов не применяется

Секция `RESULT_CACHE` не может применяться при выполнении любых из следующих условий:

- Функция определяется в разделе объявлений анонимного блока. Чтобы результаты функции могли кэшироваться, функция должна определяться на уровне схемы или внутри пакета.
- Функция является *конвейерной* табличной функцией.
- Функция имеет параметры `OUT` или `IN OUT`. В этом случае функция может только возвращать данные из секции `RETURN`.
- Какой-либо из параметров `IN` функции относится к одному из следующих типов: `BLOB`, `CLOB`, `NCLOB`, `REF CURSOR`, коллекция, запись или объектный тип.
- Возвращаемый тип функции относится к одному из следующих типов: `BLOB`, `CLOB`, `NCLOB`, `REF CURSOR`, объектный тип, коллекция или запись, содержащие какие-либо из перечисленных типов (например, коллекция `CLOB` не подойдет для кэширования результатов функции).
- Функция использует модель прав вызывающего, и вы используете Oracle Database 11g. В 11g попытка определения функции с кэшированием результатов и правами вызывающего приводит к ошибке компиляции (PLS-00999). Хорошие новости: в Oracle Database 12c это ограничение было снято, то есть стало возможным кэширование результатов функций, определенных с секцией `AUTHID CURRENT_USER`. Концептуально все выглядит так, словно Oracle передает имя пользователя в невидимом аргументе функции.
- Функция обращается к таблицам словарей данных, временным таблицам, последовательностям или недетерминированным функциям `SQL`.

Не используйте (или по крайней мере очень внимательно проанализируйте использование) секции `RESULT_CACHE` в случае истинности каких-либо из следующих условий:

- Функция имеет побочные эффекты; например, она изменяет содержимое таблиц базы данных или внешнее состояние приложения (скажем, отправляя данные в `sysout` через `DBMS_OUTPUT` или отправляя электронную почту). Так как вы не можете быть уверены в том, когда будет выполнено тело функции (и будет ли оно выполнено вообще), скорее всего, приложение не будет правильно работать во всех возможных обстоятельствах. Это слишком дорогая цена за повышение производительности.
- Ваша функция (или находящийся в ней запрос) содержит зависимости, относящиеся к сеансу, — такие, как ссылки на `SYSDATE` или `USER`, зависимости настроек `NLS` (например, вызов `TO_CHAR`, зависящий от модели форматирования по умолчанию) и т. д.

- Ваша функция выполняет запрос к таблице, находящейся под действием политики безопасности VPD (Virtual Private Database). Последствия использования VPD с кэшированием результатов функций описаны в разделе «Детализированные зависимости в версии 11.2 и выше».

Полезная информация о поведении кэширования результатов функций

Следующая информация пригодится каждому, кто захочет подробно разобраться в тонкостях кэширования результатов функций в приложениях:

- Проверая, вызывалась ли ранее функция с теми же входными данными, Oracle считает, что значение NULL равно NULL. Иначе говоря, если моя функция имеет строковый аргумент и вызывается со входным значением NULL, то при следующем вызове со значением NULL Oracle решает, что вызывать функцию не нужно, и возвращает кэшированный результат.
- Пользователи никогда не видят некорректные данные. Предположим, функция возвращает фамилию работника по идентификатору, и фамилия «Feuerstein» кэшируется с идентификатором 400. Если пользователь затем изменит содержимое таблицы `employees`, даже если это изменение еще не было зафиксировано, база данных обойдет кэш (и любой другой кэш, зависящий от `employees`) для данного сеанса. Все остальные пользователи, подключенные к экземпляру (или RAC в Oracle Database 11g Release 2 и выше), продолжают использовать кэш.
- Если функция передает необработанное исключение, база данных не кэширует входные данные этого выполнения; иначе говоря, содержимое кэша результатов для этой функции не изменится.

Управление кэшем результатов функций

Кэш результатов функций представляет собой область памяти в SGA. Oracle представляет стандартный набор средств для управления кэшем:

- `RESULT_CACHE_MAX_SIZE` *параметр инициализации* — задает максимальный объем памяти SGA, которая будет использоваться для кэша результатов функций. В случае заполнения кэша Oracle использует алгоритм вытеснения по давности использования (LRU, Least Recently Used) для вытеснения из кэша самых старых данных.
- `DBMS_RESULT_CACHE` *пакет* — пакет с набором подпрограмм для управления содержимым кэша. Пакет в основном представляет интерес для администраторов базы данных.

Динамические представления

- `V$RESULT_CACHE_STATISTICS` — различные настройки кэша результатов и статистика использования, включая размер блока и количество успешно созданных результатов в кэше.
- `V$RESULT_CACHE_OBJECTS` — все объекты, для которых были кэшированы результаты.
- `V$RESULT_CACHE_MEMORY` — все блоки памяти и их состояние с обратными ссылками на представление `V$RESULT_CACHE_OBJECTS` по столбцу `object_id`.
- `V$RESULT_CACHE_DEPENDENCY` — отношения зависимости между кэшированными результатами и зависимостями.

Процедура, которая может использоваться для вывода зависимостей:

```
/* Файл в Сети: show_frc_dependencies.sql */
CREATE OR REPLACE PROCEDURE show_frc_dependencies (
    name_like_in    IN VARCHAR2)
IS
BEGIN
    DBMS_OUTPUT.put_line ('Dependencies for "' || name_like_in || '"');

    FOR rec
```



```

    IN (SELECT d.result_id
        /* Clean up display of function name */
        , TRANSLATE (SUBSTR (res.name, 1, INSTR (res.name, ':') - 1)
            , 'A"', 'A')
            function_name
        , dep.name depends_on
    FROM v$result_cache_dependency d
        , v$result_cache_objects res
        , v$result_cache_objects dep
    WHERE res.id = d.result_id
        AND dep.id = d.depend_id
        AND res.name LIKE name_like_in)
LOOP
    /* Не включать зависимости от себя */
    IF rec.function_name <> rec.depends_on
    THEN
        DBMS_OUTPUT.put_line (
            rec.function_name || ' depends on ' || rec.depends_on);
    END IF;
END LOOP;
END;
/

```

Детализированные зависимости в версии 11.2 и выше

Существенным улучшением версии 11.2 в отношении кэширования результатов функций стало детализированное отслеживание зависимостей. Oracle теперь запоминает, от каких таблиц зависит каждый набор кэшированных данных (значения аргументов **IN** и результат). Иначе говоря, разные строки кэшированных данных могут иметь разные наборы зависимых таблиц. При фиксации изменений в таблицах Oracle удаляет из кэша только те результаты, которые зависят от этих таблиц, — не обязательно весь кэш (как произошло бы в версии 11.1).

Как правило, функции, к которым применяется кэширование результатов, зависят от одних и тех же таблиц (не считая фактических значений, передаваемых для формальных параметров). В следующем примере зависимости могут изменяться:

```

/* Файл в Сети: 11g_frc_dependencies.sql */
CREATE TABLE tablea (col VARCHAR2 (2));

CREATE TABLE tableb (col VARCHAR2 (2));

BEGIN
    INSERT INTO tablea VALUES ('a1');
    INSERT INTO tableb VALUES ('b1');
    COMMIT;
END;
/

CREATE OR REPLACE FUNCTION dynamic_query (table_suffix_in VARCHAR2)
RETURN VARCHAR2
RESULT_CACHE
IS
    l_return VARCHAR2 (2);
BEGIN
    DBMS_OUTPUT.put_line ('SELECT FROM table' || table_suffix_in);

    EXECUTE IMMEDIATE 'select col from table' || table_suffix_in INTO l_return;

    RETURN l_return;
END;
/

```

Как видите, если передать суффикс `a`, функция получает строку из таблицы `TABLEA`, а при передаче `b` функция выполняет выборку из `TABLEB`. Теперь предположим, что был выполнен следующий сценарий, который использует приведенную выше процедуру для вывода изменений в кэше результатов и его зависимостей:

```
/* Файл в Сети: 11g_frc_dependencies.sql */
DECLARE
  l_value  VARCHAR2 (2);
BEGIN
  l_value := dynamic_query ('a');
  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After a(1)');

  l_value := dynamic_query ('b');
  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After b(1)');

  UPDATE tablea SET col = 'a2';
  COMMIT;

  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After change to a2');

  l_value := dynamic_query ('a');
  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After a(2)');

  l_value := dynamic_query ('b');
  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After b(2)');

  UPDATE tableb SET col = 'b2';
  COMMIT;

  show_frc_dependencies ('%DYNAMIC_QUERY%', 'After change to b2');
END;
/
```

Результат выполнения предыдущего сценария:

```
SELECT FROM tablea
After a(1): Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEA

SELECT FROM tableb
After b(1): Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEB
HR.DYNAMIC_QUERY depends on HR.TABLEA

After change to a2: Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEB

SELECT FROM tablea
After a(2): Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEB
HR.DYNAMIC_QUERY depends on HR.TABLEA

After b(2): Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEB
HR.DYNAMIC_QUERY depends on HR.TABLEA

After change to b2: Dependencies for "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depends on HR.TABLEA
```

Как видите, даже после внесения (и фиксации) изменений в строке одной таблицы кэшированный результат функции `dynamic_query` сбрасывается не полностью. Удаляются только строки, зависящие от этой конкретной таблицы.

Виртуальные приватные базы данных и кэширование результатов функций

При использовании в приложениях виртуальных приватных баз данных, или VPD, определяются политики безопасности операций SQL с таблицами. База данных Oracle затем автоматически добавляет эти политики в форме предикатов WHERE для ограничения строк, которые пользователь может запрашивать или изменять в конкретной таблице. Обойти эти политики невозможно, так как они применяются на уровне SQL — и остаются невидимыми для пользователя. Таким образом, пользователи, подключенные к двум разным схемам, могут выполнить одинаковый (на первый взгляд) запрос (например, `SELECT last_name FROM employees`) и получить разные результаты. За подробной информацией о безопасности уровня строк обращайтесь к главе 23.

А пока рассмотрим пример тривиального использования VPD и возможности поставки некорректных данных о пользователе (весь код из этого раздела содержится в файле `11g_frc_vpd.sql` на сайте книги). Предположим, я определяю следующий пакет с двумя функциями в схеме приложения: одна функция возвращает фамилию работника с заданным идентификатором, а другая используется как политика безопасности VPD:

```
/* Файл в Сети: 11g_frc_vpd.sql */
PACKAGE emplu11g
IS
    FUNCTION last_name (employee_id_in IN employees.employee_id%TYPE)
        RETURN employees.last_name%TYPE
        result_cache;

    FUNCTION restrict_employees (schema_in VARCHAR2, NAME_IN VARCHAR2)
        RETURN VARCHAR2;
END emplu11g;

PACKAGE BODY emplu11g
IS
    FUNCTION last_name (employee_id_in IN employees.employee_id%TYPE)
        RETURN employees.last_name%TYPE
        RESULT_CACHE RELIES_ON (employees)
    IS
        onerow_rec    employees%ROWTYPE;
    BEGIN
        DBMS_OUTPUT.PUT_LINE ( 'Looking up last name for employee ID '
                                || employee_id_in );
        SELECT * INTO onerow_rec
        FROM employees
        WHERE employee_id = employee_id_in;

        RETURN onerow_rec.last_name;
    END last_name;

    FUNCTION restrict_employees (schema_in VARCHAR2, NAME_IN VARCHAR2)
        RETURN VARCHAR2
    IS
    BEGIN
        RETURN (CASE USER
                WHEN 'HR' THEN '1 = 1'
                ELSE '1 = 2'
                END
        );
    END restrict_employees;
END emplu11g;
```

Функция `restrict_employees` работает по очень простому принципу: при подключении к схеме `HR` видны все строки таблицы `employees`; в остальных случаях не видно ничего. Затем эта функция назначается политикой безопасности для всех операций с таблицей `employees`:

```
BEGIN
  DBMS_RLS.add_policy
    (object_schema      => 'HR'
    , object_name        => 'employees'
    , policy_name        => 'rls_and_rc'
    , function_schema    => 'HR'
    , policy_function    => 'emply11g.restrict_employees'
    , statement_types    => 'SELECT,UPDATE,DELETE,INSERT'
    , update_check       => TRUE
    );
END;
```

Затем схеме `SCOTT` предоставляется возможность выполнения пакета и выборки из таблицы:

```
GRANT EXECUTE ON emply11g TO scott
/
GRANT SELECT ON employees TO scott
/
```

Прежде чем выполнять функцию, убедимся в том, что политика безопасности работает и влияет на данные, видимые схемам `HR` и `SCOTT`.

Я подключаюсь к схеме `HR` и успешно запрашиваю данные из таблицы `employees`:

```
SELECT last_name
  FROM employees
 WHERE employee_id = 198/
LAST_NAME
-----
OConnell
```

Теперь тот же запрос выполняется с подключением к `SCOTT`; обратите внимание на различия!

```
CONNECT scott/tiger@oracle11
SELECT last_name
  FROM hr.employees
 WHERE employee_id = 198/
no rows selected.
```

Механизм VPD работает: при подключении к `SCOTT` я не вижу строки данных, видимые из `HR`. Теперь посмотрим, что произойдет при выполнении того же запроса из функции с кэшированием результатов, принадлежащей `HR`. Сначала я подключаюсь к схеме `HR` и выполняю функцию, после чего вывожу возвращенное имя:

```
BEGIN
  DBMS_OUTPUT.put_line (emply11g.last_name (198));END;/
Looking up last name for employee ID 198
OConnell
```

Обратите внимание на две строки выходных данных:

1. Сообщение «Looking up last name for employee ID 198» выводится потому, что функция была выполнена.
2. Строка «OConnell» выводится потому, что строка данных была найдена, а функция вернула фамилию.

Теперь я подключаюсь к схеме `SCOTT` и выполняю тот же блок кода. Так как функция выполняет команду `SELECT INTO`, которая *не должна* возвращать строки, я ожидаю увидеть необработанное исключение `NO_DATA_FOUND`. Вместо этого...

```
SQL> BEGIN
  2   DBMS_OUTPUT.put_line (hr.emplu11g.last_name (198));END;/
OConnell
```

Функция успешно возвращает «OConnell», но обратите внимание: текст «Looking up...» не выводится. Это объясняется тем, что ядро PL/SQL не выполняет эту функцию (и вызов DBMS_OUTPUT.PUT_LINE внутри функции); оно просто возвращает кэшированную фамилию.

И именно это обстоятельство делает комбинацию VPD с кэшированием результатов функций такой опасной. Так как функция сначала вызывалась с входным значением 198 от HR, фамилия, связанная с идентификатором, была кэширована для всех остальных сеансов, подключенных к тому же экземпляру. Таким образом, пользователь, подключенный к схеме SCOTT, видит данные, которые ему видеть не положено.

Чтобы убедиться в том, что при отсутствии кэширования функция действительно будет возвращать NO_DATA_FOUND, подключимся к HR и объявим содержимое кэша недействительным, зафиксировав изменение в таблице employees (подойдет любое изменение):

```
BEGIN
  UPDATE employees SET salary = salary * 1.5;
  COMMIT;END;/
```

Теперь при подключении к схеме SCOTT и выполнении функции я получаю необработанное исключение NO_DATA_FOUND:

```
SQL> BEGIN
  2   DBMS_OUTPUT.put_line (hr.emplu11g.last_name (198));END;/
ORA-01403: no data found
ORA-06512: at "HR.EMPLU11G", line 10
ORA-06512: at line 3
```

Итак, если вы работаете над одним из относительно редких приложений, использующих технологию VPD, будьте крайне осторожны с функциями, использующими кэширование результатов.

Сводка способов кэширования

Если значение не изменялось с момента его последнего запроса, постарайтесь найти способ сведения к минимуму времени его выборки. Как было доказано за многие годы на примере SGA в архитектуре баз данных Oracle, технология кэширования данных играет важнейшую роль в оптимизации производительности. Прозрачное кэширование курсоров, блоков данных и т. д. демонстрирует, как организовать собственное кэширование или использовать непрозрачные кэши SGA (что потребует внесения изменений в код).

Ниже приведена краткая сводка рекомендаций в области кэширования данных. Основные варианты:

- **Пакетное кэширование** — создание кэша пакетного уровня (скорее всего, в виде коллекции), который будет хранить ранее выбранные данные и предоставлять их из памяти PGA намного быстрее, чем при выборке из SGA. У этого способа кэширования есть два основных недостатка: данные копируются для каждого сеанса, подключенного к базе данных Oracle, и кэш не может обновляться при внесении изменений сеансом в таблицу, из которой берутся кэшированные данные.
- **Кэширование детерминированных функций** — там, где это уместно, определяйте свои функции как детерминированные (DETERMINISTIC). При наличии этого ключевого слова входные данные и возвращаемые значения функции будут кэшироваться в области действия одного запроса SQL.
- **Кэширование результатов функций** — этот механизм кэширования (Oracle Database 11g и выше) используется в том случае, если данные запрашиваются из таблицы

намного чаще, чем изменяются. Этот декларативный метод кэширования функций почти не уступает по скорости пакетному кэшированию. Содержимое кэша совместно используется всеми сеансами, подключенными к экземпляру, и может автоматически объявляться недействительным при внесении изменений в таблицы, из которых были получены кэшированные данные.

Массовая обработка

В Oracle были введены новые средства, значительно повышающие производительность запросов в PL/SQL, — команда `FORALL` и секция `BULK COLLECT`. Они объединяются общим термином *конструкций массовой обработки* (bulk processing). Для чего нужны эти конструкции, спросите вы? Как известно, язык PL/SQL тесно интегрирован с SQL-ядром базы данных Oracle. Он является основным языком программирования для Oracle, несмотря на то что теперь в базе данных также может использоваться язык Java.

Однако эта интеграция не означает, что выполнение кода SQL из программы PL/SQL не сопряжено с затратами. В ходе обработки блока программного кода ядро PL/SQL выполняет процедурные команды самостоятельно, а команды SQL передает ядру SQL. Уровень SQL выполняет команды и при необходимости возвращает результаты ядру PL/SQL. Передача управления между ядрами PL/SQL и SQL (рис. 21.2) называется *переключением контекста*. Каждое переключение контекста приводит к дополнительным затратам ресурсов. Необходимость в переключении контекста, приводящим к снижению производительности, встречается во многих ситуациях. PL/SQL и SQL тесно интегрированы на синтаксическом уровне, но во внутренней реализации они связаны отнюдь не так тесно, как кажется на первый взгляд.

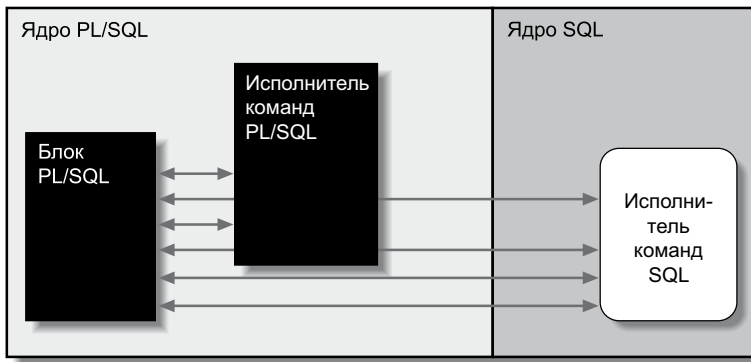


Рис. 21.2. Переключения контекста между PL/SQL и SQL

Однако конструкции `FORALL` и `BULK COLLECT` позволяют оптимизировать взаимодействие этих двух компонентов; фактически вы приказываете ядру PL/SQL объединить множество переключений контекста в одно переключение, тем самым повышая производительность своего приложения.

Рассмотрим команду `FORALL`, изображенную на рис. 21.3. Вместо того чтобы перебирать обновляемые записи в курсорном цикле `FOR` или в цикле со счетчиком, мы используем заголовок `FORALL` для определения общего количества итераций. На стадии выполнения ядро PL/SQL «расширяет» команду `UPDATE` в набор команд для выполнения всех итераций, а затем передает их ядру SQL за одно переключение контекста. Иначе говоря, выполняются те же команды SQL, но за одно обращение к ядру SQL.

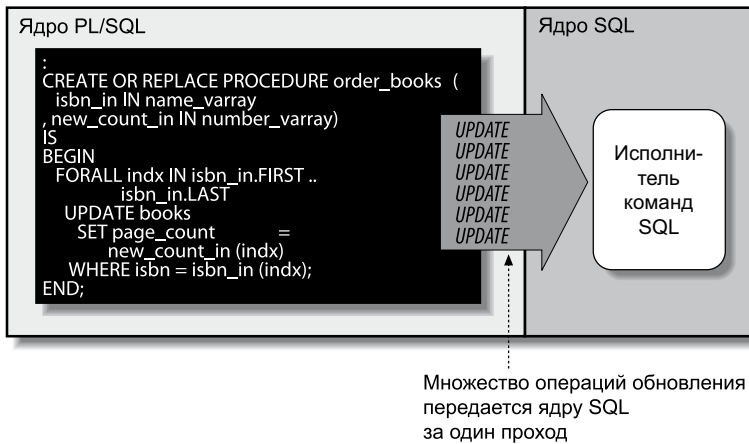


Рис. 21.3. Одно переключение контекста при использовании `FORALL`

Сокращение количества переключений контекста приводит к значительному ускорению выполнения в PL/SQL команд SQL, выполняющих выборку нескольких строк данных. Давайте поближе познакомимся с конструкциями `BULK COLLECT` и `FORALL`.

Ускорение выборки с использованием `BULK COLLECT`

Синтаксис `BULK COLLECT` позволяет за одно обращение к базе данных извлечь из явного или неявного курсора несколько строк данных. Выборка данных с помощью запроса с секцией `BULK COLLECT` сокращает количество переключений контекста между PL/SQL и SQL, благодаря чему работа выполняется быстрее и с меньшими затратами.

Рассмотрим следующий фрагмент кода. Нам нужно прочитать сотни строк данных об автомобилях, загрязняющих окружающую среду. Эти данные помещаются в набор коллекций, что значительно упрощает их дальнейший анализ и обработку:

```
DECLARE
    TYPE names_t IS TABLE OF transportation.name%TYPE;
    TYPE mileage_t IS TABLE OF transportation.mileage %TYPE;
    names names_t;
    mileages mileage_t;
    CURSOR major_polluters_cur
    IS
        SELECT name, mileage FROM transportation
        WHERE transport_type = 'AUTOMOBILE' AND mileage < 20;
BEGIN
    FOR bad_car IN major_polluters_cur
    LOOP
        names.EXTEND;
        names (major_polluters_cur %ROWCOUNT) := bad_car.NAME;
        mileages.EXTEND;
        mileages (major_polluters_cur%ROWCOUNT) := bad_car.mileage;
    END LOOP;
    -- Далее можно работать с данными в коллекциях
END;
```

Этот код выполняет поставленную задачу, но далеко не самым эффективным образом. Если, к примеру, таблица `transportation` содержит 2000 строк, PL/SQL придется выполнить 2000 операций выборки данных из глобальной системной области.

Секция `BULK COLLECT` способна значительно ускорить выборку. Включив ее в курсор (явный или неявный), вы указываете ядру SQL на необходимость перед возвратом

управления PL/SQL связать выходные данные из множества строк с заданными коллекциями. Синтаксис этой секции таков:

```
... BULK COLLECT INTO имя_коллекции[, имя_коллекции] ...
```

Здесь *имя_коллекции* определяет коллекцию, в которую должны быть помещены выходные данные курсора.

Некоторые правила и ограничения, связанные с использованием секции **BULK COLLECT**:

- До выхода Oracle9i секция **BULK COLLECT** могла использоваться только в статических командах **SQL**. Последующие версии поддерживают ее применение и в динамическом **SQL**.
- Ключевые слова **BULK COLLECT** могут использоваться только в секциях **SELECT INTO**, **FETCH INTO** и **RETURNING INTO**.
- Ядро **SQL** автоматически инициализирует и расширяет коллекции, заданные в секции **BULK COLLECT**. Заполнение начинается с индекса 1, далее элементы вставляются последовательно (без пропусков), с заменой определенных ранее элементов.
- Команда **SELECT...BULK COLLECT** не выдает исключение **NO_DATA_FOUND**, если при выборке не получено ни одной строки. Наличие данных проверяется по содержимому коллекции.
- Если запрос не вернул ни одной строки, метод **COUNT** коллекции возвращает 0.

Рассмотрим эти правила на примерах. Новая версия предыдущего примера, усовершенствованная с помощью секции **BULK COLLECT**:

```
DECLARE
  TYPE names_t IS TABLE OF transportation.name%TYPE;
  TYPE mileage_t IS TABLE OF transportation.mileage %TYPE;
  names names_t;
  mileages mileage_t;
BEGIN
  SELECT name, mileage BULK COLLECT INTO names, mileages
    FROM transportation
   WHERE transport_type = 'AUTOMOBILE'
     AND mileage < 20;
  /* Далее можно работать с данными в коллекциях */
END;
```

Из программы удаляется код инициализации и расширения коллекций.

Для решения задачи не обязательно использовать неявные курсоры — этот пример также можно переписать с использованием явного курсора:

```
DECLARE
  TYPE names_t IS TABLE OF transportation.name%TYPE;
  TYPE mileage_t IS TABLE OF transportation.mileage %TYPE;
  names names_t;
  mileages mileage_t;

  CURSOR major_polluters_cur IS
    SELECT name, mileage FROM transportation
   WHERE transport_type = 'AUTOMOBILE' AND mileage < 20;
BEGIN
  OPEN major_polluters_cur;
  FETCH major_polluters_cur BULK COLLECT INTO names, mileages;
  CLOSE major_polluters_cur;
  ...
END;
```

Я также могу упростить свою работу, выполнив выборку в коллекцию записей:

```
DECLARE
  TYPE transportation_aat IS TABLE OF transportation%ROWTYPE
    INDEX BY PLS_INTEGER;
  l_transportation transportation_aat;
```



```

BEGIN
  SELECT * BULK COLLECT INTO l_transportation
    FROM transportation
     WHERE transport_type = 'AUTOMOBILE'
       AND mileage < 20;
  -- Теперь работаем с данными в коллекциях
END;
```



В Oracle10g и последующих версиях компилятор PL/SQL автоматически оптимизирует курсорный цикл FOR, чтобы его производительность была сравнима с производительностью BULK COLLECT. Вам не нужно заниматься явным преобразованием этого кода — если только в теле цикла не выполняются (прямо или косвенно) команды DML. База данных не оптимизирует команды DML в FORALL, поэтому вы должны явно преобразовать курсорный цикл FOR для использования BULK COLLECT. После этого коллекции, заполненные с BULK COLLECT, используются для управления командой FORALL.

Ограничение на количество возвращаемых строк

Для секций BULK COLLECT Oracle поддерживает условие LIMIT, ограничивающее количество строк, выбираемых из базы данных. Его синтаксис таков:

```
FETCH курсор BULK COLLECT INTO ... [LIMIT строки];
```

Здесь *строки* — это литерал, переменная или выражение, возвращающее значение типа NUMBER (иначе иницируется исключение VALUE_ERROR).

Ограничение очень полезно при использовании BULK COLLECT, потому что оно помогает управлять объемом памяти, используемым программой при обработке данных. Допустим, вам потребовалось запросить и обработать 10 000 строк данных. Вы *можете* использовать BULK COLLECT для выборки всех записей и заполнения объемистой коллекции, однако такой подход приводит к значительным затратам памяти в PGA. Его выполнение из разных схем Oracle приведет к снижению производительности приложения из-за необходимости выгрузки PGA.

В следующем блоке секция LIMIT используется в команде FETCH, вызываемой в простом цикле:

```

DECLARE
  CURSOR allrows_cur IS SELECT * FROM employees;
  TYPE employee_aat IS TABLE OF allrows_cur%ROWTYPE
    INDEX BY BINARY_INTEGER;
  l_employees employee_aat;
BEGIN
  OPEN allrows_cur;
  LOOP
    FETCH allrows_cur BULK COLLECT INTO l_employees LIMIT 100;

    /* Обработка данных с перебором содержимого коллекции. */
    FOR l_row IN 1 .. l_employees.COUNT
      LOOP
        upgrade_employee_status (l_employees(l_row).employee_id);
      END LOOP;

    EXIT WHEN allrows_cur%NOTFOUND;
  END LOOP;

  CLOSE allrows_cur;
END;
```

Обратите внимание: цикл прерывается по проверке значения allrows_cur%NOTFOUND в конце цикла. При выборке данных по одной строке этот код обычно размещается сразу

же за командой `FETCH`. С конструкцией `BULK COLLECT` так поступать не стоит, потому что при достижении последнего набора строк курсор будет исчерпан (а `%NOTFOUND` вернет `TRUE`), но при этом в коллекции останутся элементы, которые необходимо обработать. Проверьте либо атрибут `%NOTFOUND` в конце цикла, либо содержимое коллекции непосредственно после выборки:

LOOP

```
    FETCH allrows_cur BULK COLLECT INTO l_employees LIMIT 100;
    EXIT WHEN l_employees.COUNT = 0;
```

Недостаток второго решения заключается в выполнении дополнительной выборки, не возвращающей строк (по сравнению с проверкой `%NOTFOUND` в конце тела цикла).



В Oracle Database 12c появилась возможность использования секции `FIRST ROWS` для ограничения количества строк, возвращаемых выборкой с `BULK COLLECT`. В следующем программном блоке используется команда `FETCH` с секцией `LIMIT`, которая выполняется в простом цикле. Этот код читает только первые 50 строк, определяемых командой `SELECT`:

```
DECLARE
    TYPE salaries_t IS TABLE OF employees.salary%TYPE;
    l_salaries salaries_t;
BEGIN
    SELECT salary BULK COLLECT INTO sals FROM employees
    FETCH FIRST 50 ROWS ONLY;
END;
/
```

Выборка нескольких столбцов

Как показывают приведенные примеры, секция `BULK COLLECT` позволяет выбрать из курсора данные нескольких столбцов и поместить их в разные коллекции. Несомненно, было бы элегантнее извлечь набор столбцов в одну коллекцию записей. Такая возможность появилась в Oracle9i Release 2.

Допустим, нам нужно извлечь из таблицы `transportation` информацию обо всех машинах с расходом топлива менее галлона на 20 миль. Задача решается с минимальным объемом кода:

```
DECLARE
    -- Объявление типа коллекции
    TYPE VehTab IS TABLE OF transportation%ROWTYPE;

    -- Создание экземпляра коллекции на основе типа
    gas_guzzlers VehTab;
BEGIN
    SELECT *
        BULK COLLECT INTO gas_guzzlers
        FROM transportation
        WHERE mileage < 20;
    ...
```

До Oracle9i Release 2 при выполнении этого кода было бы выдано исключение (PLS-00597). При извлечении данных из курсора в коллекцию записей также можно использовать секцию `LIMIT`, ограничивающую количество выбираемых строк.

Использование `RETURNING` с `BULK COLLECT`

Вы уже знаете, как использовать секцию `BULK COLLECT` с явными и неявными курсорами. Ее также можно включить в команду `FORALL`, чтобы воспользоваться секцией `RETURNING`.

Секция **RETURNING** позволяет получить информацию из команды **DML**. Таким образом, благодаря **RETURNING** вы обходитесь без дополнительных запросов к базе данных для определения результата только что завершенной операции **DML**.

Допустим, конгресс принял закон, по которому оклад самых высокооплачиваемых сотрудников компании не должен превышать оклад низкооплачиваемых сотрудников более чем в 50 раз. Общее число сотрудников составляет 250 000 человек. Поскольку руководство компании не намерено уменьшать доходы генерального директора, решено повысить оклады сотрудников, зарабатывающих менее 1/50 части его дохода в 145 миллионов долларов, и урезать оклады всего остального высшего руководства — должен же кто-то компенсировать расходы.

Чтобы пересчитать и обновить такое количество данных, придется изрядно потрудиться. Команда **FORALL** ускорит работу, однако после выполнения всех расчетов и внесения изменений нам нужно будет вывести отчет со старыми и новыми окладами. В этом нам пригодится секция **RETURNING**.

Начнем с функции, которая возвращает величину оклада генерального директора:

```
/* Файл в Сети: onlyfair.sql */
FUNCTION salforexec (title_in IN VARCHAR2) RETURN NUMBER
IS
    CURSOR ceo_compensation IS
        SELECT salary + bonus + stock_options +
            mercedes_benz_allowance + yacht_allowance
            FROM compensation
            WHERE title = title_in;
    big_bucks NUMBER;
BEGIN
    OPEN ceo_compensation;
    FETCH ceo_compensation INTO big_bucks;
    RETURN big_bucks;
END;
```

В главном блоке программы обновления мы объявим ряд локальных переменных и запрос для определения низкооплачиваемых и высокооплачиваемых сотрудников, чья зарплата будет обновлена:

```
DECLARE
    big_bucks NUMBER := salforexec ('CEO');
    min_sal NUMBER := big_bucks / 50;
    names name_varray;
    old_salaries number_varray;
    new_salaries number_varray;

    CURSOR affected_employees (ceosal IN NUMBER)
    IS
        SELECT name, salary + bonus old_salary
            FROM compensation
            WHERE title != 'CEO'
              AND ((salary + bonus < ceosal / 50)
                 OR (salary + bonus > ceosal / 10)) ;
```

В начале исполняемого раздела определяемые запросом данные помещаются в две коллекции командой **FETCH**, содержащей секцию **BULK COLLECT**:

```
OPEN affected_employees (big_bucks);
FETCH affected_employees
    BULK COLLECT INTO names, old_salaries;
```

Затем команда **FORALL** выполняет последовательный перебор элементов коллекции **names**:

```
FORALL indx IN names.FIRST .. names.L*
    UPDATE compensation
```

продолжение ➤

```

SET salary =
  GREATEST(
    DECODE (
      GREATEST (min_sal, salary),
      min_sal, min_sal,
      salary / 5),
    min_sal )
WHERE name = names (indx)
RETURNING salary BULK COLLECT INTO new_salaries;

```

Команда `DECODE` либо увеличивает оклад сотрудника на 80%, либо уменьшает его. В конце используется секция `RETURNING`, с помощью которой мы помещаем новые значения в коллекцию `new_salaries`.

Вот и все. Благодаря секции `RETURNING` нам не придется выполнять еще один запрос, выбирающий из базы данных новые оклады сотрудников, и можно сразу приступить к формированию отчета:

```

FOR indx IN names.FIRST .. names.LAST
LOOP
  DBMS_OUTPUT.PUT_LINE (
    RPAD (names(indx), 20) ||
    RPAD (' Old: ' || old_salaries(indx), 15) ||
    ' New: ' || new_salaries(indx)
  );
END LOOP;

```

А вот как выглядит отчет, сгенерированный сценарием `onlyfair.sql`:

| | | |
|----------------------|---------------|--------------|
| John DayAndNight | Old: 10500 | New: 2900000 |
| Holly Cubicle | Old: 52000 | New: 2900000 |
| Sandra Watchthebucks | Old: 22000000 | New: 4000000 |



Значения столбцов и выражения, возвращаемые секцией `RETURNING` команды `FORALL`, добавляются в коллекции после ранее записанных туда значений. Если же секция `RETURNING` используется в обычном цикле `FOR`, предыдущие значения заменяются данными, возвращенными последней командой `DML`.

Быстрое выполнение операций DML и команда FORALL

`BULK COLLECT` ускоряет выполнение запросов на выборку. `FORALL` делает то же самое для операций вставки, обновления, удаления и слияния (`FORALL` с командой `MERGE` поддерживается только в Oracle11g). `FORALL` приказывает ядру PL/SQL выполнить массовую привязку всех элементов одной или нескольких коллекций перед отправкой команд ядру SQL.

С учетом основополагающей роли SQL в приложениях на базе Oracle и влияния команд `DML` на общую производительность команда `FORALL`, вероятно, является самым важным средством оптимизации в языке PL/SQL.

Итак, если вы еще не используете `FORALL`, у меня есть для вас и плохие, и хорошие новости. Плохие новости: кодовая база вашего приложения годами не совершенствовалась для использования важнейших нововведений Oracle. Хорошие новости: когда вы начнете использовать `FORALL`, ваши пользователи будут радоваться очень приятному (и относительно легко достижимому) приросту производительности.

Ниже объясняются основные особенности и нюансы использования `FORALL` с множеством примеров.

Синтаксис оператора FORALL

Хотя команда FORALL выполняет итеративную обработку (то есть перебирает все строки коллекции), она не является циклом FOR, а потому не имеет ни команды LOOP, ни команды END LOOP. Ее синтаксис выглядит так:

```
FORALL индекс IN
  [ нижняя_граница ... верхняя_граница |
    INDICES OF коллекция |
    VALUES OF коллекция
  ]
  [ SAVE EXCEPTIONS ]
  команда_sql;
```

Здесь *индекс* — целочисленная переменная, неявно объявляемая Oracle; *нижняя_граница* — начальное значение индекса (строка или элемент коллекции); *верхняя_граница* — конечное значение индекса (строка или элемент коллекции); *команда_sql* — команда SQL, выполняемая для каждого элемента коллекции; *коллекция* — коллекция PL/SQL, используемая для выборки индексов в массиве, упоминаемом в *команде_sql*. Конструкции INDICES OF и VALUES OF поддерживаются начиная с Oracle10g. Необязательная секция SAVE EXCEPTIONS указывает FORALL на необходимость обработки всех строк данных с сохранением всех возникающих исключений.

При использовании FORALL необходимо соблюдать следующие правила:

- Тело команды FORALL должно представлять собой одну команду DML — INSERT, UPDATE, DELETE или MERGE (в Oracle11g и выше).
- Команда DML должна содержать ссылки на элементы коллекции, индексируемые в команде FORALL. Область видимости переменной *индекс* ограничивается командой FORALL; ссылаться на нее за пределами цикла нельзя. Помните, что *нижняя_граница* и *верхняя_граница* не обязаны задавать все множество элементов коллекции.
- Переменная цикла не должна объявляться явно. Ядро PL/SQL объявляет ее автоматически с типом PLS_INTEGER.
- *Нижняя_граница* и *верхняя_граница* должны задавать допустимый диапазон смежных индексов для коллекции, используемой в команде SQL. Для разреженных коллекций выдается сообщение об ошибке (ORA-22160).

Соответствующий пример приведен в файле `missing_element.sql` на сайте книги.

Начиная с версии Oracle Database 10g допускается использование синтаксиса INDICES OF и VALUES OF для работы с разреженными коллекциями (содержащими неопределенные элементы между FIRST и LAST). Эти конструкции рассматриваются позднее в этой главе.

- До Oracle Database 11g ссылки на поля в коллекциях записей в командах DML были запрещены. Вместо этого можно было ссылаться только на строку коллекции в целом, независимо от того, были ли ее поля коллекциями скалярных значений или коллекциями более сложных объектов. Например, следующий код:

```
DECLARE
  TYPE employee_aat IS TABLE OF employees%ROWTYPE
    INDEX BY PLS_INTEGER;
  l_employees employee_aat;
BEGIN
  FORALL l_index IN l_employees.FIRST .. l_employees.LAST
    INSERT INTO employee (employee_id, last_name)
      VALUES (l_employees (l_index).employee_id
        , l_employees (l_index).last_name
      );
END;
```

в версиях, предшествующих Oracle Database 11g, приводил к выдаче ошибки компилятора (PLS-00436).

Чтобы использовать **FORALL** в этом случае, приходилось загружать идентификаторы работников и фамилии в две разные коллекции. К счастью, в Oracle Database 11g это ограничение было снято.

- Индекс коллекции, используемый в команде DML, не мог быть выражением. Например, в следующем сценарии:

```
DECLARE
    names name_varray := name_varray ();
BEGIN
    FORALL indx IN names.FIRST .. names.LAST
        DELETE FROM emp WHERE ename = names(indx+10);
END;
```

компилятор выдавал сообщение об ошибке (PLS-00430).

Примеры использования **FORALL**

Несколько примеров использования **FORALL**:

- Изменение количества страниц для всех книг, коды ISBN которых присутствуют в коллекции `isbns_in`:

```
PROCEDURE order_books (
    isbns_in IN name_varray,
    new_counts_in IN number_varray)
IS
BEGIN
    FORALL indx IN isbns_in.FIRST .. isbns_in.LAST
        UPDATE books
            SET page_count = new_counts_in (indx)
            WHERE isbn = isbns_in (indx);
END;
```

В этом примере изменения сводятся к замене **FOR** на **FORALL**, а также удалению ключевых слов **LOOP** и **END LOOP**. В этой команде **FORALL** ядру SQL передаются все записи, определяемые в двух коллекциях. Изменения в поведении представлены на рис. 21.3.

- Следующий пример показывает, что команда DML может содержать ссылки на несколько коллекций. В данном случае используются три коллекции: `denial`, `patient_name` и `illnesses`. При этом индексируются только первые две коллекции, то есть при вызове **INSERT** передаются отдельные элементы коллекции. В третий столбец `health_coverage` для каждой вставляемой записи включается коллекция:

```
FORALL indx IN denial.FIRST .. denial.LAST
    INSERT INTO health_coverage
        VALUES (denial(indx), patient_name(indx), illnesses);
```

- Использование секции **RETURNING** в команде **FORALL** для получения информации о каждой отдельной операции **DELETE**. Обратите внимание на необходимость использования **BULK COLLECT INTO** в секции **RETURNING**:

```
FUNCTION remove_emps_by_dept (deptlist IN dlist_t)
    RETURN enolist_t
IS
    enolist enolist_t;
BEGIN
    FORALL aDept IN deptlist.FIRST..deptlist.LAST
        DELETE FROM employees WHERE department_id IN deptlist(aDept)
            RETURNING employee_id BULK COLLECT INTO enolist;
    RETURN enolist;
END;
```

- Использование индексов, определяемых в одной коллекции, для определения строк данных коллекции, которые должны использоваться динамической командой **INSERT**:

```
FORALL indx IN INDICES OF l_top_employees
EXECUTE IMMEDIATE
  'INSERT INTO ' || l_table || ' VALUES (:emp_pky, :new_salary)'
  USING l_new_salaries(indx).employee_id,
        l_new_salaries(indx).salary;
```

Атрибуты курсоров для FORALL

Вы можете использовать атрибуты курсоров после выполнения команды **FORALL** для получения информации об операции DML, выполняемой в **FORALL**. Oracle также предоставляет дополнительный атрибут **%BULK_ROWCOUNT** для получения более детализированной информации о результатах массовой команды DML.

В табл. 21.1 описаны значения, возвращаемые этими атрибутами для **FORALL**.

Таблица 21.1. Неявные атрибуты курсоров для команд **FORALL**

| Имя | Описание |
|---------------------|---|
| SQL%FOUND | Возвращает TRUE, если при последнем выполнении команды SQL была модифицирована хотя бы одна строка |
| SQL%NOTFOUND | Возвращает TRUE, если команда DML не модифицировала ни одной строки |
| SQL%ROWCOUNT | Возвращает количество строк, модифицированных командой DML |
| SQL%ISOPEN | Всегда возвращает FALSE (не используется) |
| SQL%BULK_ROWCOUNT | Возвращает псевдоколлекцию, которая сообщает количество строк, обработанных каждой соответствующей командой SQL, выполненной через FORALL . Обратите внимание: если значение %BULK_ROWCOUNT(j) равно нулю, то атрибуты %FOUND и %NOTFOUND будут равны FALSE и TRUE соответственно |
| SQL%BULK_EXCEPTIONS | Возвращает псевдоколлекцию с информацией обо всех исключениях, инициализированных в команде FORALL с секцией SAVE EXCEPTIONS . |

Составной атрибут **%BULK_ROWCOUNT**, созданный специально для **FORALL**, обладает семантикой ассоциативного массива или коллекции. База данных помещает в N-й элемент коллекции количество строк, обработанных при N-м выполнении **INSERT**, **UPDATE**, **DELETE** или **MERGE** команды **FORALL**. Если операция не затронула ни одной строки, N-я строка содержит нуль.

Пример использования **%BULK_ROWCOUNT** (а также общего атрибута **%ROWCOUNT**):

```
DECLARE
  TYPE isbn_list IS TABLE OF VARCHAR2(13);
  my_books isbn_list
  := isbn_list (
    '1-56592-375-8', '0-596-00121-5', '1-56592-849-0',
    '1-56592-335-9', '1-56592-674-9', '1-56592-675-7',
    '0-596-00180-0', '1-56592-457-6'
  );
BEGIN
  FORALL book_index IN
    my_books.FIRST..my_books.LAST
  UPDATE books
    SET page_count = page_count / 2
    WHERE isbn = my_books (book_index);

  -- Было ли обновлено ожидаемое число строк?
  IF SQL%ROWCOUNT != 8
  THEN
    DBMS_OUTPUT.PUT_LINE (
```

продолжение ➤

```

        'We are missing a book!');
END IF;

-- 4-я команда UPDATE изменила какие-либо строки?
IF SQL%BULK_ROWCOUNT(4) = 0
THEN
    DBMS_OUTPUT.PUT_LINE (
        'What happened to Oracle PL/SQL Programming?');
END IF;
END;
```

Несколько замечаний по поводу работы этого атрибута:

Команды **FORALL** и **%BULK_ROWCOUNT** используют одинаковые значения индексов или номера строк коллекций. Например, если коллекция, переданная **FORALL**, содержит данные в строках с 10 по 200, то псевдоколлекция **%BULK_ROWCOUNT** также будет содержать определенные и заполненные строки с 10 по 200. Все остальные строки будут неопределенными.

Если **INSERT** влияет только на одну строку (например, при указании списка **VALUES**), значение строки в **%BULK_ROWCOUNT** будет равно 1. При этом для команд **INSERT...SELECT** значение **%BULK_ROWCOUNT** может быть больше 1.

Значение строки в псевдомассиве **%BULK_ROWCOUNT** для операций удаления, обновления и вставки с выборкой может быть любым натуральным числом (0 или положительным); эти команды могут изменять более одной строки в зависимости от их условий **WHERE**.

Поведение ROLLBACK для FORALL

Команда **FORALL** позволяет передать ядру SQL несколько команд SQL. Это означает, что переключение контекста всего одно — но каждая команда выполняется ядром SQL отдельно от других.

Что случится, если в одной из этих команд SQL произойдет ошибка?

1. Команда DML, инициировавшая исключение, откатывается от неявной точки сохранения, созданной ядром PL/SQL перед выполнением команды. Изменения во всех строках, модифицированных этой командой, отменяются.
2. Все предшествующие операции DML в этой команде **FORALL**, уже завершённые без ошибок, не отменяются.
3. Если вы не приняли специальных мер (добавив секцию **SAVE EXCEPTIONS** в **FORALL** — см. далее), выполнение **FORALL** останавливается, и остальные команды вообще не выполняются.

Продолжение после исключений и секция SAVE EXCEPTIONS

Добавляя в заголовок **FORALL** секцию **SAVE EXCEPTIONS**, вы приказываете Oracle продолжить обработку даже при возникновении ошибки. База данных «сохраняет исключение» (или несколько исключений, если ошибок было несколько). При завершении команды DML инициируется исключение **ORA-24381**. Далее в разделе исключений можно обратиться к псевдоколлекции **SQL%BULK_EXCEPTIONS** для получения информации об ошибке.

Пример с пояснениями:

```

/* Файл в Сети: bulkexc.sql */
1 DECLARE
2     bulk_errors    EXCEPTION;
3     PRAGMA EXCEPTION_INIT (bulk_errors, -24381);
4     TYPE namelist_t IS TABLE OF VARCHAR2(32767);
5
6     enames_with_errors  namelist_t
7         := namelist_t ('ABC',
```



```

8          'DEF',
9          NULL, /* Фамилия должна быть отлична от NULL */
10         'LITTLE',
11         RPAD ('BIGBIGGERBIGGEST', 250, 'ABC'), /* Слишком длинное */
12         'SMITHIE'
13     );
14 BEGIN
15     FORALL indx IN enames_with_errors.FIRST .. enames_with_errors.LAST
16         SAVE EXCEPTIONS
17         UPDATE EMPLOYEES
18             SET last_name = enames_with_errors (indx);
19
20 EXCEPTION
21     WHEN bulk_errors
22     THEN
23         DBMS_OUTPUT.put_line ('Updated ' || SQL%ROWCOUNT || ' rows.');
```

```

24
25     FOR indx IN 1 .. SQL%BULK_EXCEPTIONS.COUNT
26     LOOP
27         DBMS_OUTPUT.PUT_LINE ('Error '
28             || indx
29             || ' occurred during '
30             || 'iteration '
31             || SQL%BULK_EXCEPTIONS (indx).ERROR_INDEX
32             || ' updating name to '
33             || enames_with_errors (SQL%BULK_EXCEPTIONS (indx).ERROR_INDEX);
34         DBMS_OUTPUT.PUT_LINE ('Oracle error is '
35             || SQLERRM ( -1 * SQL%BULK_EXCEPTIONS (indx).ERROR_CODE)
36         );
37     END LOOP;
38 END;
```

Если выполнить этот код в режиме SERVEROUTPUT, будут получены следующие результаты:

```
SQL> EXEC bulk_exceptions
```

```
Error 1 occurred during iteration 3 updating name to BIGBIGGERBIGGEST
Oracle error is ORA-01407: cannot update () to NULL
```

```
Error 2 occurred during iteration 5 updating name to
Oracle error is ORA-01401: inserted value too large for column
```

Другими словами, база данных обнаружила два исключения при обработке команд DML для коллекции. Она не прервала выполнение при первом исключении, а продолжила работу и зарегистрировала второе исключение.

Функциональность обработки ошибок в этом коде описана в следующей таблице.

| Строки | Описание |
|---------|---|
| 2–3 | Объявление именованного исключения для удобства чтения раздела исключений |
| 4–13 | Объявление и заполнение коллекции, управляющей выполнением FORALL. В коллекцию намеренно включены два элемента, для которых будут инициированы исключения |
| 15–18 | Выполнение команды UPDATE для FORALL с использованием коллекции enames_with_errors |
| 25–37 | Использование числового цикла FOR для перебора содержимого псевдоколлекции SQL%BULK_EXCEPTIONS. Обратите внимание: я могу вызывать метод COUNT для получения количества определенных строк (инициированных ошибок), но не могу вызывать другие методы — такие, как FIRST и LAST |
| 31 и 33 | Поле ERROR_INDEX каждой строки псевдоколлекции возвращает номер строки в управляющей коллекции команды FORALL, в которой произошло исключение |
| 35 | Поле ERROR_CODE каждой строки псевдоколлекции возвращает номер ошибки для инициированного исключения. Следует учесть, что значение хранится в виде положительного целого числа; прежде чем передавать его SQLERRM или выводить информацию, необходимо умножить его на -1 |

Управление FORALL для непоследовательных массивов

До выхода Oracle Database 10g коллекция, используемая в команде FORALL, должна была быть плотной или последовательно заполненной. При наличии пропусков между первым и последним значениями в диапазоне из заголовка FORALL, как в следующем коде, выдавалась ошибка:

```

1  DECLARE
2      TYPE employee_aat IS TABLE OF employees.employee_id%TYPE
3          INDEX BY PLS_INTEGER;
4      l_employees    employee_aat;
5  BEGIN
6      l_employees (1) := 100;
7      l_employees (100) := 1000;
8      FORALL l_index IN l_employees.FIRST .. l_employees.LAST
9          UPDATE employees SET salary = 10000
10             WHERE employee_id = l_employees (l_index);
11  END;
12  /

```

Сообщение об ошибке выглядит так:

```

DECLARE
*
ERROR at line 1:
ORA-22160: element at index [2] does not exist

```

Кроме того, в массиве нельзя пропустить те строки, которые не должны обрабатываться командой FORALL. Эти ограничения часто приводят к написанию дополнительного кода сжатия коллекций, чтобы те удовлетворяли ограничениям FORALL. Чтобы избавить разработчиков от этой раздражающей необходимости, в Oracle Database 10g язык PL/SQL был дополнен конструкциями INDICES OF и VALUES OF для задания части массива, которая должна обрабатываться FORALL.

Начнем с различий между этими секциями, а затем я приведу пару примеров, демонстрирующих их применение.

INDICES OF — эта секция используется в том случае, если у вас имеется коллекция (назовем ее *индексным массивом*), строки которой определяют, какие строки основного массива (из команды DML FORALL) должны обрабатываться. Иначе говоря, если элемент в позиции N не определен в индексном массиве, то команда FORALL должна проигнорировать элемент в позиции N основного массива.

VALUES OF — секция используется в том случае, если у вас имеется коллекция целых чисел (также называемая индексным массивом), содержимое которой (значения элементов) определяет обрабатываемые позиции основного массива.

Пример использования INDICES OF

Допустим, я хочу обновить оклады некоторых работников значением 10 000. В настоящее время таких значений в таблице нет:

```

SQL> SELECT employee_id FROM employees WHERE salary = 10000;
no rows selected

```

Я пишу следующую программу:

```

/* Файл в Сети: 10g_indices_of.sql */
1  DECLARE
2      TYPE employee_aat IS TABLE OF employees.employee_id%TYPE
3          INDEX BY PLS_INTEGER;
4
5      l_employees    employee_aat;
6

```

```

7      TYPE boolean_aat IS TABLE OF BOOLEAN
8      INDEX BY PLS_INTEGER;
9
10     l_employee_indices boolean_aat;
11 BEGIN
12     l_employees (1) := 7839;
13     l_employees (100) := 7654;
14     l_employees (500) := 7950;
15     --
16     l_employee_indices (1) := TRUE;
17     l_employee_indices (500) := TRUE;
18     l_employee_indices (799) := TRUE;
19
20     FORALL l_index IN INDICES OF l_employee_indices
21         BETWEEN 1 AND 500
22             UPDATE employees23 SET salary = 10000
23             WHERE employee_id = l_employees (l_index);
25 END;
```

Логика программы кратко описана в следующей таблице.

| Строки | Описание |
|--------|--|
| 2–5 | Определение коллекции с идентификаторами работников |
| 7–10 | Определение коллекции логических значений |
| 12–14 | Заполнение (разреженное) трех строк (1, 100 и 500) в коллекции идентификаторов работников |
| 16–18 | В коллекции определяются только две строки, 1 и 500 |
| 20–24 | В команде FORALL вместо того, чтобы указывать диапазон значений между FIRST и LAST, я просто указываю INDICES OF l_employee_indices. Также включается дополнительная секция BETWEEN для ограничения используемых значений индексов |

После выполнения кода я выдаю запрос, который показывает, что в таблице были обновлены всего две строки — строка работника с идентификатором 7654 была пропущена, потому что в позиции 100 таблицы индексов элемент не определен:

```
SQL> SELECT employee_id FROM employee WHERE salary = 10000;
```

```

EMPLOYEE_ID
-----
       7839
       7950
```

С конструкцией INDICES OF (строка 20) *содержимое* индексного массива игнорируется. Важны лишь позиции или номера строк, определенных в коллекции.

Пример использования VALUES OF

В этом примере, как и в предыдущем, оклад некоторых работников обновляется значением 10 000 — на этот раз с секцией VALUES OF. В данный момент строк с таким значением в таблице нет:

```
SQL> SELECT employee_id FROM employee WHERE salary = 10000;
no rows selected
```

Программа выглядит так:

```

/* Файл в Сети: 10g_values_of.sql */
1  DECLARE
2      TYPE employee_aat IS TABLE OF employees.employee_id%TYPE
3      INDEX BY PLS_INTEGER;
4
5      l_employees          employee_aat;
6
7      TYPE indices_aat IS TABLE OF PLS_INTEGER
```

продолжение ➤

```

8      INDEX BY PLS_INTEGER;
9
10     l_employee_indices  indices_aat;
11 BEGIN
12     l_employees (-77) := 7820;
13     l_employees (13067) := 7799;
14     l_employees (99999999) := 7369;
15     --
16     l_employee_indices (100) := -77;
17     l_employee_indices (200) := 99999999;
18     --
19     FORALL l_index IN VALUES OF l_employee_indices
20         UPDATE employees
21             SET salary = 10000
22             WHERE employee_id = l_employees (l_index);
23 END;
```

Логика программы кратко описана в следующей таблице.

| Строки | Описание |
|--------|---|
| 2–6 | Определение коллекции с идентификаторами работников |
| 7–10 | Определение коллекции целых чисел |
| 12–14 | Заполнение (разреженное) трех строк (–77, 13067 и 99999999) в коллекции идентификаторов работников |
| 16–17 | Индексный массив идентифицирует строки, используемые при обновлении. Так как я использую VALUES OF, номера строк несущественны — важны значения, хранящиеся в каждой строке индексного массива. «Средняя» строка 13067 должна быть пропущена, поэтому я определяю в массиве l_employee_indices всего две строки, которым присваиваются значения –77 и 99999999 соответственно |
| 19–22 | Вместо того чтобы задавать диапазон значений от FIRST до LAST, я просто включаю конструкцию VALUES OF l_employee_indices. Обратите внимание: в коллекции индексов заполняются строки 100 и 200. VALUES OF не требует плотного заполнения индексной коллекции |

После выполнения кода я снова выдаю запрос, который показывает, что в таблице были обновлены всего две строки — строка работника с идентификатором 7799 была пропущена, потому что «коллекция значений» не содержит элемента, значение которого равно 13067:

```
SQL> SELECT employee_id FROM employees WHERE salary = 10000;

EMPLOYEE_ID
-----
          7369
          7820
```

Повышение производительности с использованием конвейерных табличных функций

Конвейерные функции сочетают элегантность и простоту PL/SQL с производительностью SQL. Разработка и поддержка сложных преобразований данных не требует значительных усилий в PL/SQL, но для достижения высокопроизводительной обработки данных приходится часто прибегать к решениям на базе SQL. Конвейерные функции заполняют пробел между двумя технологиями, но у них также имеются свои уникальные особенности, которые делают их превосходным инструментом оптимизации.

В этом разделе будут приведены примеры типичных требований к обработке данных и их оптимизации с применением конвейерных функций. Мы рассмотрим следующие темы:

- Адаптация типичных требований к загрузке данных для конвейерных функций. В каждом случае старые решения, ориентированные на строки, заменяются решениями, ориентированными на наборы, использующими параллельные конвейерные функции.
- Использование параллельного контекста конвейерных функций для повышения производительности выгрузки данных.
- Относительная производительность группировки и потоковой передачи для параллельных конвейерных функций.
- Обработка конвейерных и стандартных табличных функций оптимизатором CBO (Cost-Based Optimizer).
- Решение проблем сложной многотабличной загрузки с применением конвейерных функций.

Основной синтаксис конвейерных табличных функций был рассмотрен в главе 17. Напомню, что конвейерные функции вызываются в секции `FROM` команд `SQL` и используются для получения данных, как если бы они были реляционными таблицами или другими источниками строк. В отличие от стандартных табличных функций, которые должны полностью обработать свои данные, прежде чем передавать коллекцию данных (возможно, достаточно большую) на сторону контекста вызова, конвейерные табличные функции поставляют свои результаты клиенту почти сразу же после их подготовки. Другими словами, конвейерные функции не материализуют весь результирующий набор, и эта оптимизация существенно сокращает затраты памяти PGA. Другая отличительная особенность конвейерных функций — возможность их вызова в контексте параллельного запроса. Я много раз использовал эти средства для повышения эффективности своих программ; далее я покажу, когда и как вы можете воспользоваться ими в своих программах.

Замена вставки на базе строк загрузкой на базе конвейерных функций

Для демонстрации возможностей конвейерных функций представим типичный сценарий, который нам нужно довести до современного уровня. На базе примера `stockpivot` я запрограммировал простую построчную загрузку с выборкой исходных данных `stockpivot` и преобразованием каждой записи в две строки для вставки. Программа содержится в пакете и выглядит так:

```
/* Файл в Сети: stockpivot_setup.sql */
PROCEDURE load_stocks_legacy IS

    CURSOR c_source_data IS
        SELECT ticker, open_price, close_price, trade_date
        FROM   stocktable;

    r_source_data stockpivot_pkg.stocktable_rt;
    r_target_data stockpivot_pkg.tickertable_rt;

BEGIN
    OPEN c_source_data;
    LOOP
        FETCH c_source_data INTO r_source_data;
        EXIT WHEN c_source_data%NOTFOUND;

        /* Цена на момент открытия... */
        r_target_data.ticker      := r_source_data.ticker;
        r_target_data.price_type := '0';
```

продолжение ➤

```

r_target_data.price      := r_source_data.open_price;
r_target_data.price_date := r_source_data.trade_date;
INSERT INTO tickertable VALUES r_target_data;

/* Цена на момент закрытия... */
r_target_data.price_type := 'C';
r_target_data.price      := r_source_data.close_price;
INSERT INTO tickertable VALUES r_target_data;

END LOOP;
CLOSE c_source_data;
END load_stocks_legacy;
```

Я регулярно встречаю подобный код, и еще со времен Oracle8i Database обычно использую BULK COLLECT и FORALL в качестве основного средства оптимизации (когда логика слишком сложна для решения на базе SQL). Альтернативное решение (описание которого я впервые увидел у Тома Кайта¹) основано на использовании вставки на базе наборов из конвейерной функции. Другими словами, конвейерная функция используется для всех преобразований данных и подготовительной логики, но загрузка данных в целевую таблицу реализуется отдельно через вставку из набора. Освоив этот мощный прием, я успешно использую его для оптимизации своих проектов.

Реализация конвейерной функции

Как было показано в главе 17, создание конвейерной функции следует начать с анализа данных, которые она должна возвращать. Для этого необходимо создать объектный тип, определяющий одну строку возвращаемых данных конвейерной функции:

```

/* Файл в Сети: stockpivot_setup.sql */
CREATE TYPE stockpivot_ot AS OBJECT
( ticker      VARCHAR2(10)
, price_type  VARCHAR2(1)
, price       NUMBER
, price_date  DATE
);
```

Также необходимо создать коллекцию таких объектов, определяющих возвращаемый тип функции:

```

/* Файл в Сети: stockpivot_setup.sql */
CREATE TYPE stockpivot_ntt AS TABLE OF stockpivot_ot;
```

Преобразовать унаследованный код в конвейерную функцию несложно. Работа начинается с определения спецификации функции в заголовке. Также необходимо включить процедуру загрузки, которая будет описана позднее:

```

/* Файл в Сети: stockpivot_setup.sql */
CREATE PACKAGE stockpivot_pkg AS

    TYPE stocktable_rct IS REF CURSOR
        RETURN stocktable%ROWTYPE;

    <...>

    FUNCTION pipe_stocks(
        p_source_data IN stockpivot_pkg.stocktable_rct
    ) RETURN stockpivot_ntt PIPELINED;

    PROCEDURE load_stocks;

END stockpivot_pkg;
```

¹ См. «Expert Oracle Database Architecture» (издательство Apress), с. 640–643.

Моя конвейерная функция получает сильную ссылку REF CURSOR (впрочем, в данной ситуации также можно использовать слабую ссылку). Параметр-курсор не является обязательным — с таким же успехом его можно было объявить в самой функции (как было сделано в старой процедуре). Впрочем, параметр-курсор все равно понадобится в следующих итерациях конвейерной функции, поэтому я добавил его.

Реализация функции выглядит так:

```

/* Файл в Сети: stockpivot_setup.sql */
1  FUNCTION pipe_stocks(
2      p_source_data IN stockpivot_pkg.stocktable_rct
3      ) RETURN stockpivot_ntt PIPELINED IS
4
5      r_target_data stockpivot_ot := stockpivot_ot(NULL, NULL, NULL, NULL);
6      r_source_data stockpivot_pkg.stocktable_rt;
7
8  BEGIN
9      LOOP
10         FETCH p_source_data INTO r_source_data;
11         EXIT WHEN p_source_data%NOTFOUND;
12
13         /* Первая строка... */
14         r_target_data.ticker      := r_source_data.ticker;
15         r_target_data.price_type := 'O';
16         r_target_data.price      := r_source_data.open_price;
17         r_target_data.price_date := r_source_data.trade_date;
18         PIPE ROW (r_target_data);
19
20         /* Вторая строка... */
21         r_target_data.price_type := 'C';
22         r_target_data.price      := r_source_data.close_price;
23         PIPE ROW (r_target_data);
24
25     END LOOP;
26     CLOSE p_source_data;
27     RETURN;
28 END pipe_stocks;

```

Кроме общего синтаксиса конвейерных функций (с которым вы уже знакомы по главе 17), основная часть кода конвейерной функции легко узнается по старой версии. Основные различия перечислены в следующей таблице.

| Строки | Описание |
|---------|---|
| 2 | Старая версия курсора удаляется из кода; вместо нее передается параметр CURSOR |
| 5 | Целевая переменная теперь определяется не с типом ROWTYPE таблицы, а с объектным типом STOCKPIVOT_OT, определяющим возвращаемые данные конвейерной функции |
| 18 и 23 | Вместо того чтобы вставлять записи в таблицу, я передаю записи из функции. На этой стадии база данных буферизует небольшое количество конвейерных объектных строк в соответствующую коллекцию. В зависимости от размера массива клиента эта буферизованная коллекция данных становится доступной практически немедленно |

Загрузка данных из конвейерной функции

Как видите, при небольшом количестве изменений в исходном коде я создал конвейерную функцию, которая подготавливает и передает все данные для загрузки в tickertable. Чтобы завершить преобразование старого кода, остается написать дополнительную процедуру для вставки данных в таблицу:

```

/* Файл в Сети: stockpivot_setup.sql */
PROCEDURE load_stocks IS

```

продолжение ➞

BEGIN

```
INSERT INTO tickertable (ticker, price_type, price, price_date)
SELECT ticker, price_type, price, price_date
FROM TABLE(
    stockpivot_pkg.pipe_stocks(
        CURSOR(SELECT * FROM stocktable)));
```

END load_stocks;

На этом преобразование построчного кода в решение с конвейерной функцией можно считать законченным. Как оно работает в сравнении с оригиналом? В процессе тестирования я создал `stocktable` как внешнюю таблицу в виде файла из 500 000 записей. Старая версия с построчным преобразованием завершалась за 57 секунд (со вставкой 1 миллиона записей в `tickertable`), а вставка с конвейерной функцией выполнялась всего за 16 секунд (результаты тестирования для всех примеров доступны на сайте книги).

Если учесть, что это была самая первая и простая реализация конвейерной функции, улучшение производительности выглядит вполне достойно. Тем не менее оно все же уступает производительности простого решения с `BULK COLLECT` и `FORALL` (которое в моих тестах обрабатывало всего за 5 секунд), поэтому в конвейерную функцию загрузки данных необходимо внести кое-какие изменения.

Но перед этим обратите внимание на то, что я сохранил построчную выборку из основного курсора и ничего не сделал для сокращения «дорогостоящего» переключения контекста (для чего потребуется выборка `BULK COLLECT`). Почему же новая версия работает быстрее старой?

Прежде всего из-за того, что операции DML с наборами (такие, как `INSERT...SELECT` в моей реализации) почти всегда выполняются намного быстрее, чем построчные процедурные решения. В этом конкретном случае выигрыш был обусловлен внутренней оптимизацией вставок с наборами. А именно при вставке `INSERT...SELECT` база данных сохраняет намного меньше данных отмены, чем для обычной вставки (`INSERT...VALUES`). Другими словами, при вставке 100 строк в одной команде будет сгенерировано меньше информации отмены, чем при вставке 100 строк по одной строке.

Для моей исходной таблицы с 1 миллионом строк объем данных отмены составил свыше 270 Мбайт. При конвейерной загрузке он сократился до 37 Мбайт — согласитесь, весьма заметная экономия!



В своем примере я для наглядности отказался от сложных преобразований данных. Всегда следует предполагать, что правила обработки данных достаточно сложны, чтобы для их реализации потребовалось решение с конвейерной функцией PL/SQL. В противном случае для преобразования данных большого объема было бы проще воспользоваться решением на базе SQL с аналитическими функциями, подзапросами и выражениями CASE.

Оптимизация конвейерных функций посредством выборки из массива

Старый код был усовершенствован реализацией на базе конвейерной функции, но работа еще не закончена. Есть и другие возможности оптимизации, и обработка должна по крайней мере не уступать по скорости решению с `BULK COLLECT` и `FORALL`. Обратите внимание: в моем решении используется построчная выборка из основного курсора. Следовательно, первая возможность для оптимизации — использование массовой выборки `BULK COLLECT`.

Для начала я добавляю в свою спецификацию пакета размер массива по умолчанию. Оптимальный размер массива зависит от конкретных требований к обработки данных, но я всегда начинаю тестирование со 100, а дальше действую по ситуации. Также в спецификацию пакета будет добавлен тип ассоциативного массива (впрочем, его с таким же успехом можно было объявить в теле) для массовой выборки из курсора. Наконец, я добавляю в сигнатуру конвейерной функции второй параметр для управления размером выборки (конечно, это не обязательно — просто полезная привычка). Теперь спецификация выглядит так:

```
/* Файл в Сети: stockpivot_setup.sql */
CREATE PACKAGE stockpivot_pkg AS
<...>
  c_default_limit CONSTANT PLS_INTEGER := 100;

  TYPE stocktable_aat IS TABLE OF stocktable%ROWTYPE
    INDEX BY PLS_INTEGER;

  FUNCTION pipe_stocks_array(
    p_source_data IN stockpivot_pkg.stocktable_rct,
    p_limit_size  IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit
  ) RETURN stockpivot_ntt PIPELINED;
<...>
END stockpivot_pkg;
```

Сама функция очень похожа на исходную версию:

```
/* Файл в Сети: stockpivot_setup.sql */
FUNCTION pipe_stocks_array(
  p_source_data IN stockpivot_pkg.stocktable_rct,
  p_limit_size  IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit
) RETURN stockpivot_ntt PIPELINED IS

  r_target_data stockpivot_ot := stockpivot_ot(NULL, NULL, NULL, NULL);
  aa_source_data stockpivot_pkg.stocktable_aat;

BEGIN
  LOOP
    FETCH p_source_data BULK COLLECT INTO aa_source_data LIMIT p_limit_size;
    EXIT WHEN aa_source_data.COUNT = 0;

    /* Обработка пакета из (p_limit_size) записей... */
    FOR i IN 1 .. aa_source_data.COUNT LOOP
      /* Первая строка ... */
      r_target_data.ticker      := aa_source_data(i).ticker;
      r_target_data.price_type := 'O';
      r_target_data.price      := aa_source_data(i).open_price;
      r_target_data.price_date := aa_source_data(i).trade_date;
      PIPE ROW (r_target_data);

      /* Вторая строка... */
      r_target_data.price_type := 'C';
      r_target_data.price      := aa_source_data(i).close_price;
      PIPE ROW (r_target_data);
    END LOOP;
  END LOOP;
  CLOSE p_source_data;
  RETURN;
END pipe_stocks_array;
```

Единственное отличие от исходной версии — использование **BULK COLLECT...LIMIT**. Процедура загрузки осталась прежней, только теперь она ссылается на версию конвейерной функции с массивом. Время загрузки сокращается до 6 секунд исключительно за счет сокращения переключения контекста из PL/SQL. Теперь решение с конвейерной функцией по производительности сравнимо с решением на базе **BULK COLLECT** и **FORALL**.

Использование параллельных конвейерных функций для достижения оптимальной производительности

Переход от вставки к конвейерной функции уже обеспечил неплохой выигрыш по быстродействию. Тем не менее остается еще одна возможность, которая повысит производительность по сравнению с любым другим решением: использование параллельной поддержки конвейерных функций, упоминавшейся в главе 17. В следующей итерации я включаю параллельную поддержку для функции `stockpivot`, для чего в сигнатуру функции добавляется еще одна секция:

```
/* Файл в Сети: stockpivot_setup.sql */
CREATE PACKAGE stockpivot_pkg AS
<...>
FUNCTION pipe_stocks_parallel(
    p_source_data IN stockpivot_pkg.stocktable_rct
    p_limit_size IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit
) RETURN stockpivot_ntt
    PIPELINED
    PARALLEL_ENABLE (PARTITION p_source_data BY ANY);
<...>
END stockpivot_pkg;
```

Используя схему группировки `ANY`, я приказываю базе данных Oracle случайным образом распределять исходные данные между параллельными процессами. Это связано с тем, что порядок получения и обработки исходных данных функцией не влияет на результат (то есть межстрочные зависимости отсутствуют). Конечно, так бывает не всегда.

Включение параллельного выполнения конвейерных функций

Помимо синтаксиса включения параллельного выполнения в спецификации и теле реализации функции не отличается от предыдущего примера (полный код пакета содержится в файле `stockpivot_setup.sql` на сайте). Однако я должен принять меры к тому, чтобы загрузка из `tickertable` выполнялась параллельно. Сначала нужно включить параллельные операции DML на уровне сеанса. Когда это будет сделано, параллельный запрос вызывается одним из следующих способов:

- С рекомендацией `PARALLEL`.
- С параллельными настройками `DEGREE` для используемых объектов.
- Форсированный параллельный запрос (`ALTER SESSION FORCE PARALLEL (QUERY) PARALLEL n`).



Параллельные запросы/DML относятся к функциональности Oracle Database Enterprise Edition. Если вы используете издание Standard Edition или Standard Edition One, вы не сможете использовать параллельный режим конвейерных функций.

В своей процедуре я включил параллельный режим DML на уровне сеанса и использовал рекомендации для задания степени параллелизма (DOP) 4:

```
/* Файл в Сети: stockpivot_setup.sql */
PROCEDURE load_stocks_parallel IS
BEGIN
    EXECUTE IMMEDIATE 'ALTER SESSION ENABLE PARALLEL DML';
    INSERT /*+ PARALLEL(t, 4) */ INTO tickertable t
        (ticker, price_type, price, price_date)
    SELECT ticker, price_type, price, price_date
    FROM TABLE(
        stockpivot_pkg.pipe_stocks_parallel(
            CURSOR(SELECT /*+ PARALLEL(s, 4) */ * FROM stocktable s)));
END load_stocks_parallel;
```

Время загрузки сокращается до 3 секунд — существенное улучшение по сравнению с исходной версией, а также всеми остальными версиями конвейерной функции. Конечно, при относительно быстрых операциях затраты на запуск параллельных процессов отразятся на общем времени, но несмотря на это мне удалось добиться почти 50% ускорения. Тот факт, что параллельные вставки используют метод прямой вставки (вместо традиционной) также означает, что генерируемые данные отмены снова уменьшаются — теперь они занимают всего 25 Кбайт!

В коммерческих системах приходится оптимизировать процессы, которые выполняются по часу и более, поэтому выигрыш от параллельных конвейерных операций будет значительным как в относительном, так и в абсолютном выражении.



При использовании параллельных конвейерных функций курсор-источник должен передаваться в параметре REF CURSOR. В последовательных конвейерных функциях курсор-источник может быть встроен в саму функцию (хотя я в своих примерах этого не сделал). Кроме того, для функций с группировкой по типу ANY ссылка REF CURSOR может быть как слабо, так и сильно типизированной, но с группировкой HASH или RANGE допускается только сильная типизация (за дополнительной информацией о REF CURSOR и курсорных переменных обращайтесь к главе 15).

Конвейерные функции при оптимизации операций слияния

Как видите, последовательные или параллельные конвейерные функции могут серьезно рассматриваться как механизм оптимизации крупномасштабной загрузки данных. Тем не менее загрузка не всегда подразумевает вставку, как в примере `stockpivot`. Многие загрузки данных выполняются в пошаговом режиме и требуют периодического слияния новых и измененных данных. К счастью, принцип объединения преобразований PL/SQL с наборами SQL также применим и к операциям слияния (и обновления).

Традиционное слияние в коде PL/SQL

Следующая процедура взята из примера `employee_pkg`. Требуется включить в базу данных большое количество записей `employee`, но в унаследованном коде используется старый метод PL/SQL — программа сначала пытается выполнить обновление и выполняет вставку только в том случае, если операция обновления не обнаружила ни одной подходящей записи в целевой таблице:

```
/* Файл в Сети: employees_merge_setup.sql */
PROCEDURE upsert_employees IS
    n PLS_INTEGER := 0;
BEGIN
    FOR r_emp IN (SELECT * FROM employees_staging) LOOP
        UPDATE employees
        SET     <...>
        WHERE  employee_id = r_emp.employee_id;
        IF SQL%ROWCOUNT = 0 THEN
            INSERT INTO employees (<...>)
            VALUES (<...>);
        END IF;
    END LOOP;
END upsert_employees;
```

Часть кода была удалена для краткости, но методика «обновления со вставкой» видна достаточно четко. Обратите внимание на использование неявного цикла FOR со счетчиком, который выигрывает от оптимизации выборки из массива, введенной в PL/SQL в Oracle Database 10g.

Для тестирования этой процедуры я создал громадную таблицу из 500 000 записей, после чего вставил 250 000 из них в таблицу работников, чтобы обеспечить равномерное разбиение между обновлением и вставкой. Это «примитивное слияние» на базе PL/SQL завершилось за 46 секунд.

Использование конвейерных функций для слияния с наборами

Преобразование этого примера в операцию SQL MERGE с конвейерной функцией также не вызывает особых проблем. Сначала я создаю типы вспомогательных объектов и вложенных таблиц (за подробностями обращайтесь к файлу `employees_merge_setup.sql`), затем объявляю функцию в заголовке пакета:

```
/* Файл в Сети: employees_merge_setup.sql */
CREATE PACKAGE employee_pkg AS

    c_default_limit CONSTANT PLS_INTEGER := 100;

    TYPE employee_rct IS REF CURSOR RETURN employees_staging%ROWTYPE;
    TYPE employee_aat IS TABLE OF employees_staging%ROWTYPE
        INDEX BY PLS_INTEGER;
    <...>

    FUNCTION pipe_employees(
        p_source_data IN employee_pkg.employee_rct
        p_limit_size   IN PLS_INTEGER DEFAULT employee_pkg.c_default_limit
    ) RETURN employee_ntt
        PIPELINED
        PARALLEL_ENABLE (PARTITION p_source_data BY ANY);
END employee_pkg;
```

Я включаю для конвейерной функции поддержку параллельного режима и использую схему группировки ANY, как и прежде. Реализация функции выглядит так:

```
/* Файл в Сети: employees_merge_setup.sql */
FUNCTION pipe_employees(
    p_source_data IN employee_pkg.employee_rct,
    p_limit_size  IN PLS_INTEGER DEFAULT employee_pkg.c_default_limit
) RETURN employee_ntt
    PIPELINED
    PARALLEL_ENABLE (PARTITION p_source_data BY ANY) IS
    aa_source_data employee_pkg.employee_aat;
BEGIN
    LOOP
        FETCH p_source_data BULK COLLECT INTO aa_source_data LIMIT p_limit_size;
        EXIT WHEN aa_source_data.COUNT = 0;
        FOR i IN 1 .. aa_source_data.COUNT LOOP
            PIPE ROW (
                employee_ot( aa_source_data(i).employee_id,
                            <snip>
                            SYSDATE ));
        END LOOP;
    END LOOP;
    CLOSE p_source_data;
    RETURN;
END pipe_employees;
```

Функция просто извлекает исходные данные в массив и передает их в правильном формате. Теперь ее можно использовать в команде MERGE, упакованной в процедуру пакета `employee_pkg`:

```
/* Файл в Сети: employees_merge_setup.sql */
PROCEDURE merge_employees IS
```

```

BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION ENABLE PARALLEL DML';
  MERGE /*+ PARALLEL(e, 4) */
    INTO employees e
    USING TABLE(
      employee_pkg.pipe_employees(
        CURSOR(SELECT /*+ PARALLEL(es, 4) */ *
          FROM employees_staging es))) s
    ON (e.employee_id = s.employee_id)
  WHEN MATCHED THEN
    UPDATE
      SET <snip>
  WHEN NOT MATCHED THEN
    INSERT ( <snip> )
    VALUES ( <snip> );
END merge_employees;

```

Команда SQL MERGE с параллельной конвейерной функцией сокращает время загрузки более чем на 50% до 21 секунды. Безусловно, использование параллельных конвейерных функций как источника строк для операций SQL может стать полезным инструментом оптимизации крупномасштабной загрузки данных.

Параллельные конвейерные функции при асинхронной выгрузке данных

До настоящего момента я продемонстрировал два типа загрузки данных, для которых переход на параллельную конвейерную функцию принес очевидную пользу. Параллельным режимом конвейерных функций также можно воспользоваться для выгрузки данных (я еще не видел ни одной крупной корпоративной системы хранения данных, которая бы не извлекала данные для передачи в другие системы).

Типичная программа извлечения данных

Представьте следующую ситуацию: я ежедневно извлекаю все данные своей биржевой торговли (храняемые в таблице tickertable) для передачи в контрольное подразделение, которое рассчитывает получить неструктурированный файл с разделителями. Для этого я пишу простую программу выгрузки данных из курсора:

```

/* Файл в Сети: parallel_unload_setup.sql */
PROCEDURE legacy_unload(
  p_source      IN SYS_REFCURSOR,
  p_filename    IN VARCHAR2,
  p_directory   IN VARCHAR2,
  p_limit_size  IN PLS_INTEGER DEFAULT unload_pkg.c_default_limit
) IS
  TYPE row_aat IS TABLE OF VARCHAR2(32767)
    INDEX BY PLS_INTEGER;
  aa_rows row_aat;
  v_name  VARCHAR2(128) := p_filename || '.txt';
  v_file  UTL_FILE.FILE_TYPE;
BEGIN
  v_file := UTL_FILE.FOPEN( p_directory, v_name, 'w', c_maxline );
  LOOP
    FETCH p_source BULK COLLECT INTO aa_rows LIMIT p_limit_size;
    EXIT WHEN aa_rows.COUNT = 0;
    FOR i IN 1 .. aa_rows.COUNT LOOP
      UTL_FILE.PUT_LINE(v_file, aa_rows(i));
    END LOOP;
  END LOOP;
  CLOSE p_source;
  UTL_FILE.FCLOSE(v_file);
END legacy_unload;

```

Я просто перебираю курсор, переданный в параметре, используя выборку в массив с размером 100, и записываю каждую группу записей в приемный файл с использованием UTL_FILE. Курсор-источник имеет всего один столбец — при подготовке курсора исходные столбцы объединяются конкатенацией с разделителями.

В ходе тестирования выгрузка 1 миллиона строк tickertable в неструктурированный файл заняла всего 24 секунды (я заранее несколько раз просканировал tickertable, чтобы снизить влияние физического ввода/вывода). Однако средняя длина строки tickertable составляла всего 25 байт, поэтому выгрузка происходила очень быстро. Коммерческие системы записывают существенно больший объем данных (как по длине строки, так и по числу строк), и выгрузка может занять десятки минут.

Выгрузка данных с использованием параллельной конвейерной функции

Если этот сценарий вам знаком по вашим системам, рассмотрите возможность оптимизации за счет использования параллельных конвейерных функций. Если проанализировать приведенный пример старого кода, все манипуляции с данными можно разместить в конвейерной функции (так как операции DML отсутствуют). Так почему бы не взять логику выборки из курсора и операции с UTL_FILE и не поместить их в параллельную конвейерную функцию? Это позволит существенно ускорить выгрузку данных в несколько файлов за счет применения параллельных запросов Oracle.

Конечно, конвейерные функции обычно возвращают данные в специфическом порядке, но в данном случае строки записываются в файл, и возвращать их клиенту не нужно. Вместо этого я возвращаю по одной строке на параллельный процесс с простейшими метаданными для описания сеансовой информации и количества извлеченных строк. При этом используются следующие вспомогательные типы:

```
/* Файл в Сети: parallel_unload_setup.sql */
CREATE TYPE unload_ot AS OBJECT
( file_name VARCHAR2(128)
, no_records NUMBER
, session_id NUMBER );
CREATE TYPE unload_ntt AS TABLE OF unload_ot;
```

Моя реализация базируется на унаследованной обработке с добавлением некоторой дополнительной настройки, необходимой для возвращения метаданных:

```
/* Файл в Сети: parallel_unload_setup.sql */
1  FUNCTION parallel_unload(
2      p_source      IN SYS_REFCURSOR,
3      p_filename    IN VARCHAR2,
4      p_directory   IN VARCHAR2,
5      p_limit_size  IN PLS_INTEGER DEFAULT unload_pkg.c_default_limit
6  )
7      RETURN unload_ntt
8      PIPELINED PARALLEL_ENABLE (PARTITION p_source BY ANY) AS
9      aa_rows row_aat;
10     v_sid   NUMBER := SYS_CONTEXT('USERENV','SID');
11     v_name  VARCHAR2(128) := p_filename || '_' || v_sid || '.txt';
12     v_file  UTL_FILE.FILE_TYPE;
13     v_lines PLS_INTEGER;
14 BEGIN
15     v_file := UTL_FILE.FOPEN(p_directory, v_name, 'w', c_maxline);
16     LOOP
17         FETCH p_source BULK COLLECT INTO aa_rows LIMIT p_limit_size;
18         EXIT WHEN aa_rows.COUNT = 0;
19         FOR i IN 1 .. aa_rows.COUNT LOOP
```

```

20          UTL_FILE.PUT_LINE(v_file, aa_rows(i));
21      END LOOP;
22  END LOOP;
23  v_lines := p_source%ROWCOUNT;
24  CLOSE p_source;
25  UTL_FILE.FCLOSE(v_file);
26  PIPE ROW (unload_ot(v_name, v_lines, v_sid));
27  RETURN;
28  END parallel_unload;

```

Ключевые аспекты кода этой функции представлены в следующей таблице.

| Строки | Описание |
|---------------|---|
| 1 и 8 | Функция использует параллельный режим и осуществляет группировку исходных данных по ANY. Следовательно, я могу объявить свой курсор-источник на базе системного типа SYS_REFCURSOR |
| 10 | Возвращаемые метаданные включают идентификатор сеанса (SID), который берется из контекста приложения USERENV. До Oracle Database 10g SID можно получить из таких представлений, как V\$MYSTAT |
| 11 | Выгрузка должна осуществляться параллельно в несколько файлов, поэтому для каждого параллельного вызова создается уникальное имя файла |
| 15–22 и 24–25 | Я использую всю логику обработки из исходной реализации |
| 26 | Для каждого вызова функции передается одна строка с именем файла, количеством извлеченных строк и идентификатором сеанса |

Я с минимальными усилиями включаю параллельный режим выгрузки данных, используя конвейерную функцию как асинхронный механизм разбиения. Теперь посмотрим, как вызвать новую версию:

```

/* Файл в Сети: parallel_unload_test.sql */
SELECT *
FROM TABLE(
    unload_pkg.parallel_unload(
        p_source => CURSOR(SELECT /*+ PARALLEL(t, 4) */
                                ticker      || ',' ||
                                price_type || ',' ||
                                price       || ',' ||
                                TO_CHAR(price_date, 'YYYYMMDDHH24MISS')
                                FROM tickertable t),
        p_filename => 'tickertable',
        p_directory => 'DIR' ));

```

Тестовый вывод из SQL*Plus:

| FILE_NAME | NO_RECORDS | SESSION_ID |
|---------------------|------------|------------|
| tickertable_144.txt | 260788 | 144 |
| tickertable_142.txt | 252342 | 142 |
| tickertable_127.txt | 233765 | 127 |
| tickertable_112.txt | 253105 | 112 |

4 rows selected.

Elapsed: 00:00:12.21

В моей тестовой системе с четырьмя параллельными процессами время обработки сократилось примерно вдвое. Помните, что для небольших интервалов, как в этом примере, затраты на инициализацию параллельного запуска могут повлиять на время обработки. Если извлечение данных занимает несколько минут и более, потенциальная экономия может быть существенно выше.



Этот способ оптимизации можно и дальше улучшить, «оптимизируя» вызовы `UTL_FILE` через механизм буферизации. Пример реализации приведен в функции `PARALLEL_UNLOAD_BUFFERED` в файле `parallel_unload_setup.sql` на сайте книги. Вместо того чтобы выводить каждую строку в файл немедленно, я присоединяю строки к большому буферу `VARCHAR2` (также можно было использовать коллекцию) и периодически сбрасывать его содержимое в файл. Сокращение количества вызовов `UTL_FILE` в моей системе уменьшило время выгрузки почти наполовину — всего до 7 секунд.

Влияние группировки и режима потоковой передачи в параллельных конвейерных функциях

Все примеры параллельных конвейерных функций, приводившиеся до настоящего момента, использовали схему группировки `ANY`, потому что между строками исходных данных нет зависимостей. Как упоминается в главе 17, некоторые параметры группировки и потоковой передачи управляют способом распределения входных данных и их упорядочения в параллельных процессах. Напомню:

- Параметры группировки (для распределения данных по параллельным процессам):

```
PARTITION p_cursor BY ANY
PARTITION p_cursor BY RANGE(столбцы_курсора)
PARTITION p_cursor BY HASH(столбцы_курсора)
```

- Параметры потоковой передачи (для упорядочения данных в параллельном процессе):

```
CLUSTER p_cursor BY (столбцы_курсора)
ORDER p_cursor BY (столбцы_курсора)
```

Выбор режима зависит от требований к обработке данных. Например, если вы должны позаботиться о том, чтобы все заказы конкретного клиента обрабатывались вместе, но в хронологическом порядке, используйте группировку `HASH` с режимом `ORDER`. Чтобы все данные обрабатывались в порядке наступления событий, используйте комбинацию `RANGE/ORDER`.

Относительная производительность разных комбинаций

Характеристики производительности этих режимов зависят от предполагаемого порядка сортировки. В следующей таблице приведены данные о времени, необходимом для передачи 1 миллиона строк `tickertable` через параллельную конвейерную функцию (с `DOP 4`) с использованием разных параметров группировки и потоковой передачи¹.

| Режим группировки | Режим потоковой передачи | Затраченное время |
|-------------------|--------------------------|-------------------|
| ANY | — | 5.37 |
| ANY | ORDER | 8.06 |
| ANY | CLUSTER | 9.58 |
| HASH | — | 7.48 |
| HASH | ORDER | 7.84 |
| HASH | CLUSTER | 8.10 |
| RANGE | — | 9.84 |
| RANGE | ORDER | 10.59 |
| RANGE | CLUSTER | 10.90 |

¹ Чтобы самостоятельно протестировать эти параметры, используйте файлы `parallel_options_*.sql` с сайта книги.

Как и следовало ожидать, группировки **ANY** и **HASH** показывают сравнимые результаты (хотя неупорядоченный режим **ANY** явно оказывается самым быстрым), но механизм **RANGE** значительно медленнее. Это вполне логично, потому что исходные данные должны быть упорядочены перед тем, как база данных сможет распределить их. В самих параллельных процессах упорядочение выполняется быстрее, чем группировка для всех режимов (возможно, кого-то этот результат удивит, потому что группировке не нужно упорядочивать весь набор данных), Впрочем, в вашем случае результаты могут быть иными.

Группировка с асимметричными данными

Еще один фактор, который следует принять во внимание при группировке — распределение нагрузки между параллельными процессами. Режимы **ANY** и **HASH** приводят к относительно равномерному распределению данных между параллельными процессами независимо от количества строк в источнике. Тем не менее в зависимости от характеристик данных режим **RANGE** может привести к неравномерному распределению, особенно если значения столбцов разбиения асимметричны. Если один параллельный процесс получает слишком большую часть данных, это может свести к нулю все преимущества параллельных конвейерных функций.



Во всех моих вызовах конвейерных функций передается параметр **REF CURSOR**, получаемый при помощи функции **CURSOR(SELECT...)**. Альтернативное решение — подготовка переменной **REF CURSOR** с использованием конструкции **OPEN ref_cursor FOR...** и передача этой переменной вместо вызова **CURSOR(SELECT...)**. Если вы выберете этот вариант, остерегайтесь ошибки 5349930! При использовании параллельных конвейерных функций эта ошибка может привести к неожиданному аварийному завершению параллельного процесса с исключением **ORA-01008**.

Конвейерные функции и затратный оптимизатор

Примеры этой главы демонстрируют применение конвейерных функций как простых источников, генерирующих данные для сценариев загрузки и выгрузки. Тем не менее в какой-то момент вам может потребоваться объединить конвейерную функцию с другим источником строк (например, таблица, представление или промежуточный вывод других объединений в плане выполнения **SQL**). Статистика источника строк (мощность, распределение данных, **NULL** и т. д.) играет важную роль для построения эффективного плана выполнения, но в случае конвейерных функций (а на самом деле любых табличных функций) у затратного оптимизатора (**СВО**, **Cost-Based Optimizer**) не слишком много информации для работы.

Эвристика мощности для конвейерных табличных функций

В Oracle Database 11g Release 1 и более ранних версиях затратный оптимизатор применяет эвристику *мощности* (*cardinality*) к конвейерным и табличным функциям в командах **SQL**, что иногда приводит к неэффективным планам выполнения. Мощность по умолчанию зависит от значения параметра инициализации **DB_BLOCK_SIZE**, но для базы данных со стандартным размером блока 8 Кбайт Oracle использует эвристику 8168 строк. Это легко продемонстрировать при помощи конвейерной функции, которая передает подмножество столбцов из таблицы **employees**. Используя **Autotrace** в **SQL*Plus** для генерирования плана выполнения, я получаю следующий результат:

```
/* Файл в Сети: cbo_setup.sql и cbo_test.sql */
SQL> SELECT *
      2 FROM TABLE(pipe_employees) e;
```

продолжение ➤

Execution Plan

Plan hash value: 1802204150

| Id | Operation | Name | Rows |
|----|-----------------------------------|----------------|------|
| 0 | SELECT STATEMENT | | 8168 |
| 1 | COLLECTION ITERATOR PICKLER FETCH | PIPE_EMPLOYEES | |

Конвейерная функция возвращает 50 000 строк, поэтому при объединении функции с таблицей `departments` возникает риск получить неоптимальный план:

/* Файл в Сети: cbo_test.sql */

```
SQL> SELECT *
  2 FROM   departments          d
  3 ,      TABLE(pipe_employees) e
  4 WHERE  d.department_id = e.department_id;
```

Execution Plan

Plan hash value: 4098497386

| Id | Operation | Name | Rows |
|-----|-----------------------------------|----------------|------|
| 0 | SELECT STATEMENT | | 8168 |
| 1 | MERGE JOIN | | 8168 |
| 2 | TABLE ACCESS BY INDEX ROWID | DEPARTMENTS | 27 |
| 3 | INDEX FULL SCAN | DEPT_ID_PK | 27 |
| * 4 | SORT JOIN | | 8168 |
| 5 | COLLECTION ITERATOR PICKLER FETCH | PIPE_EMPLOYEES | |

Как прогнозировалось выше, такой план выполнения наверняка будет неоптимальным; вряд ли объединение сортировкой со слиянием будет более эффективным, чем хеш-объединение в этом сценарии. Итак, как же повлиять на работу оптимизатора? В данном примере я могу использовать простые рекомендации (такие, как `LEADING` и `USE_HASH` для переопределения решения, принятого оптимизатором на основании затрат) и обеспечить хеш-объединение между таблицей и конвейерной функцией. Однако для более сложных команд SQL трудно предоставить все рекомендации, необходимые для «фиксации» плана выполнения. Часто бывает лучше предоставить СВО более полную статистику для принятия решений. Это можно сделать двумя способами:

- **Динамическая выборка оптимизатора** — функция была усовершенствована в Oracle Database 11g (11.1.0.7), и в нее была включена поддержка проб для таблиц и конвейерных функций.
- **Пользовательская оценка мощности** — существует несколько способов передачи оптимизатору оценки мощности конвейерной функции.

Оба метода будут продемонстрированы для функции `pipe_employees`.

Использование динамической выборки конвейерных функций

Динамическая выборка — чрезвычайно полезная функция, которая позволяет оптимизатору сформировать небольшую статическую выборку из одного или нескольких объектов в фазе разбора. Можете использовать динамическую выборку, если вы не собрали статистику по всем таблицам запроса или же при использовании временных объектов (например, глобальных временных таблиц). Начиная с версии 11.1.0.7 база данных Oracle может использовать динамическую выборку для таблиц или конвейерных функций.

Чтобы увидеть, на что влияет эта возможность, я повторю предыдущий запрос с включением рекомендации `DYNAMIC_SAMPLING` для функции `pipe_employees`:

```
/* Файл в Сети: cbo_test.sql */
SQL> SELECT /*+ DYNAMIC_SAMPLING(e 5) */
2      *
3  FROM  departments          d
4        , TABLE(pipe_employees) e
5  WHERE d.department_id = e.department_id;
```

Execution Plan

Plan hash value: 815920909

| Id | Operation | Name | Rows |
|-----|-----------------------------------|----------------|-------|
| 0 | SELECT STATEMENT | | 50000 |
| * 1 | HASH JOIN | | 50000 |
| 2 | TABLE ACCESS FULL | DEPARTMENTS | 27 |
| 3 | COLLECTION ITERATOR PICKLER FETCH | PIPE_EMPLOYEES | |

На этот раз СВО правильно вычисляет 50 000 строк, которые возвращает моя функция, и генерирует более подходящий план. Обратите внимание: я использую термин «вычисляет», а не «оценивает». Дело в том, что в версии 11.1.0.7 и выше оптимизатор осуществляет стопроцентную выборку данных таблицы или конвейерной функции независимо от используемого уровня динамической выборки. Я использовал уровень 5, но с таким же успехом мог использовать любой уровень от 2 до 10 — результат остался бы прежним. Конечно, это означает, что динамическая выборка может быть сопряжена с высокими затратами вычислительных ресурсов или времени, если она применяется в запросах с большими объемами данных или долго выполняемыми конвейерными функциями.

Передача статистики мощности оптимизатору

Единственная информация, которую можно явно передать оптимизатору для конвейерной функции, — это мощность. Как это часто бывает в Oracle, это можно сделать несколькими способами:

- **Рекомендация `CARDINALITY`** (не документирована) — сообщает базе данных Oracle мощность источника строк в плане выполнения. Возможность применения и эффективность этой рекомендации весьма ограничены.
- **Рекомендация `OPT_ESTIMATE`** (не документирована) — предоставляет масштабный коэффициент для корректировки оценки мощности источника данных, объединения или индекса в плане выполнения. Эта рекомендация используется в *профилях SQL* — отдельно лицензируемой функции Oracle Database 10g Enterprise Edition. Профили SQL используются для хранения масштабных коэффициентов существующих команд SQL для улучшения и стабилизации их планов выполнения.
- **Интерфейс `Extensible Optimizer`** — связывает конвейерную или табличную функцию с типом объекта для вычисления его мощности и предоставляет эту информацию непосредственно оптимизатору (доступен начиная с Oracle Database 10g).

Официально Oracle Corporation не поддерживает рекомендации `CARDINALITY` и `OPT_ESTIMATE`. По этой причине я стараюсь не использовать их в рабочем коде. Кроме профилей SQL (или динамической выборки, о которой говорилось ранее), единственным официально поддерживаемым методом передачи оценок мощности конвейерных функций оптимизатору является функциональность расширения оптимизатора, появившаяся в Oracle Database 10g.

Extensible Optimizer и мощность конвейерной функции

Расширение оптимизатора является частью Oracle-реализации *Data Cartridge* — набора интерфейсов, позволяющих расширять встроенную функциональность базы данных пользовательским кодом и алгоритмами (обычно хранящимися в объектных типах). Для конвейерных и табличных функций база данных предоставляет интерфейс, предназначенный специально для оценок мощности. В следующем простом примере для функции `pipe_employees` я связываю конвейерную функцию с особым объектным типом, который передает СВО информацию о мощности функции. Спецификация функции `pipe_employees` выглядит так:

```
/* Файл в Сети: cbo_setup.sql */
FUNCTION pipe_employees(
    p_cardinality IN INTEGER DEFAULT 1
) RETURN employee_ntt PIPELINED
```

Обратите внимание на параметр `p_cardinality`. В теле `pipe_employees` этот параметр не используется; в нем я буду сообщать СВО количество строк, которое, как ожидается, будет возвращать моя функция. А поскольку для Extensible Optimizer это должно делаться через интерфейсный тип, сначала я создам спецификацию типа объектного типа для интерфейса:

```
/* Файл в Сети: cbo_setup.sql */
1 CREATE TYPE pipelined_stats_ot AS OBJECT (
2
3     dummy INTEGER,
4
5     STATIC FUNCTION ODCIGetInterfaces (
6         p_interfaces OUT SYS.ODCIObjectList
7     ) RETURN NUMBER,
8
9     STATIC FUNCTION ODCIStatsTableFunction (
10        p_function    IN SYS.ODCIFuncInfo,
11        p_stats       OUT SYS.ODCITabFuncStats,
12        p_args        IN SYS.ODCIArgDescList,
13        p_cardinality IN INTEGER
14    ) RETURN NUMBER
15 );
```

Важные моменты этой спецификации отмечены в следующей таблице.

| Строки | Описание |
|--------|--|
| 3 | Все объектные типы должны иметь хотя бы один атрибут, поэтому я включил фиктивный атрибут с именем <code>dummy</code> , хотя он и не нужен в этом примере |
| 5 и 9 | Методы являются частью интерфейса Extensible Optimizer. Также доступно несколько других методов, но эти два необходимы для реализации интерфейса мощности для моей конвейерной функции |
| 10–12 | Эти параметры <code>ODCIStatsTableFunction</code> являются обязательными. Позиции и типы данных параметров жестко фиксированы (в отличие от их имен) |
| 13 | Все параметры конвейерной или табличной функции должны быть повторены в ассоциированном статистическом типе. В моем примере <code>pipe_employees</code> имеет единственный параметр <code>p_cardinality</code> , который также должен быть включен в сигнатуру <code>ODCIStatsTableFunction</code> |

Мой алгоритм мощности реализуется в теле типа следующим образом:

```
/* Файл в Сети: cbo_setup.sql */
1 CREATE TYPE BODY pipelined_stats_ot AS
2
3     STATIC FUNCTION ODCIGetInterfaces (
4         p_interfaces OUT SYS.ODCIObjectList
5     ) RETURN NUMBER IS
```

```

6      BEGIN
7          p_interfaces := SYS.ODCIObjectList(
8              SYS.ODCIObject ('SYS', 'ODCISTATS2')
9          );
10         RETURN ODCIConst.success;
11     END ODCIGetInterfaces;
12
13     STATIC FUNCTION ODCIStatsTableFunction (
14         p_function      IN SYS.ODCIFuncInfo,
15         p_stats          OUT SYS.ODCITabFuncStats,
16         p_args           IN SYS.ODCIArgDescList,
17         p_cardinality    IN INTEGER
18     ) RETURN NUMBER IS
19     BEGIN
20         p_stats := SYS.ODCITabFuncStats(NULL);
21         p_stats.num_rows := p_cardinality;
22         RETURN ODCIConst.success;
23     END ODCIStatsTableFunction;
24
25 END;
```

Эта реализация интерфейса очень проста; ключевые ее аспекты перечислены в следующей таблице.

| Строки | Описание |
|--------|---|
| 3–11 | Обязательное присваивание, необходимое для баз данных Oracle. Никакая пользовательская логика здесь не нужна |
| 20–21 | Мой алгоритм мощности. Для передачи СВО информации о мощности функции используется OUT-параметр p_stats. Статистический тип будет содержать ссылку на значение, переданное в параметре p_cardinality моей функции pipe_employees. Во время оптимизации запроса СВО вызовет метод ODCIStatsTableFunction для получения значения параметра p_stats и использует его в своих вычислениях |

Итак, у меня имеется конвейерная функция и статистический тип. Остается лишь связать два объекта командой SQL ASSOCIATE STATISTICS. Именно эта связь делает возможным только что описанное «волшебство»:

```

/* Файл в Сети: cbo_test.sql */
ASSOCIATE STATISTICS WITH FUNCTIONS pipe_employees USING pipelined_stats_ot;
```

Теперь все готово к тестированию. Я повторю предыдущий запрос со включением количества строк, которое, как ожидается, вернет моя конвейерная функция (50 000):

```

/* Файл в Сети: cbo_test.sql */
SQL> SELECT *
2   FROM departments d
3   ,   TABLE(pipe_employees(50000)) e
4  WHERE d.department_id = e.department_id;
```

Execution Plan

Plan hash value: 815920909

| Id | Operation | Name | Rows |
|-----|-----------------------------------|----------------|-------|
| 0 | SELECT STATEMENT | | 50000 |
| * 1 | HASH JOIN | | 50000 |
| 2 | TABLE ACCESS FULL | DEPARTMENTS | 27 |
| 3 | COLLECTION ITERATOR PICKLER FETCH | PIPE_EMPLOYEES | |

На этот раз СВО получает и использует предполагаемую мощность, и план выполнения выглядит именно так, как я ожидал. Мне даже не пришлось использовать рекомендации!

В большинстве случаев при точных входных данных СВО принимает хорошие решения, как доказывает этот пример. Конечно, мы также видим в действии «волшебство» Extensible Optimizer. Я передаю предполагаемую мощность в параметре функции `pipe_employees`, а в фазе оптимизации база данных обращается к этому параметру через сопутствующий статистический тип и использует его для правильного назначения мощности источника записей (с использованием моего алгоритма). На мой взгляд, это довольно впечатляющая возможность.

И последнее: вам определенно стоит выбрать свой способ систематической передачи информации о мощности конвейерных функций. Я продемонстрировал лишь одну из возможностей — собственно, мне следовало бы добавить параметр `p_cardinality` во *все* конвейерные функции и связать их с интерфейсным типом `pipelined_statistics_ot`. Алгоритмы, используемые в интерфейсных типах, могут быть настолько сложными, насколько потребуется. Они могут зависеть от других параметров функций (например, вы можете возвращать разные мощности в зависимости от значений параметров). А может, вы предпочтете хранить предполагаемые значения мощности в таблице, к которой интерфейсный тип будет обращаться с запросом. Помните, что эту возможность можно использовать многими разными способами.

Конвейерные функции при загрузке сложных данных

В моем примере `stockpivot` каждая входная строка преобразовывалась в две выходные строки с такой же структурой записи. Остальные примеры передавали одну выходную строку с неизменной структурой записи. Но преобразования и загрузка не всегда проходят так просто. Очень часто из одной промежуточной таблицы данные загружаются сразу в несколько таблиц — могут ли конвейерные функции пригодиться и в такой ситуации?

Да, могут; многотабличная загрузка также может оптимизироваться при помощи конвейерных функций. Сама функция может передавать столько разных типов записей, сколько потребуется, и условные или безусловные многотабличные вставки могут заполнять соответствующие таблицы нужными атрибутами.

Один источник, два приемника

Рассмотрим пример загрузки данных клиентов и адресов из одного файла. Предположим, одна запись клиента может содержать до трех адресов. Это означает, что для каждого клиента генерируется до четырех записей, например:

| CUSTOMER_ID | LAST_NAME | ADDRESS_ID | STREET_ADDRESS | PRIMARY |
|-------------|-----------|------------|------------------------------|---------|
| 1060 | Kelley | 60455 | 7310 Breathing Street | Y |
| 1060 | Kelley | 119885 | 7310 Breathing Street | N |
| 103317 | Anderson | 65045 | 57 Aguadilla Drive | Y |
| 103317 | Anderson | 65518 | 117 North Union Avenue | N |
| 103317 | Anderson | 61112 | 27 South Las Vegas Boulevard | N |

Лишние подробности удалены, но из этого примера видно, что у Келли в системе зарегистрированы два адреса, а у Андерсона — три. В моем сценарии загрузки данных для каждого клиента в таблицу клиентов должна добавляться одна запись, а в таблицу адресов — все записи адресов, которые связаны с клиентом.

Передача нескольких типов записей из конвейерных функций

Как конвейерная функция может одновременно генерировать записи клиентов и адресов? Как ни странно, у этой задачи есть два относительно простых решения:

- **Используйте заменяемые объектные типы** (см. главу 26). Функция может выдавать разные подтипы вместо супертипа, для которого написана функция. Таким образом, каждая передаваемая запись будет вставляться в соответствующую таблицу в условной многотабличной команде `INSERT FIRST`.
- **Используйте широкие, денормализованные записи** со всеми атрибутами всех целевых таблиц, хранящимися в одной передаваемой записи. Каждая передаваемая запись может преобразовываться для нескольким целевых таблиц со вставкой через многотабличную команду `INSERT ALL`.

Объектно-реляционные возможности

Начнем с первого метода, поскольку он обеспечивает самое элегантное решение. Сначала необходимо создать четыре типа для описания данных:

- «Супертип», возглавляющий иерархию типов. Супертип содержит только те атрибуты, которые должны наследоваться подтипами (в нашем примере только `customer_id`).
- Тип коллекции с элементами супертипа. Я буду использовать его как возвращаемый тип конвейерной функции.
- «Подтип» объекта клиента с остальными атрибутами, необходимыми для загрузки таблицы клиентов.
- «Подтип» объекта адреса с остальными атрибутами, необходимыми для загрузки таблицы адресов.

Для демонстрационных целей я ограничился небольшим количеством атрибутов. В нашем примере будут использоваться следующие типы:

```
/* Файл в Сети: multitype_setup.sql */
-- Супертип...
CREATE TYPE customer_ot AS OBJECT
( customer_id NUMBER
) NOT FINAL;

-- Коллекция объектов супертипа...
CREATE TYPE customer_ntt AS TABLE OF customer_ot;

-- Подтип с информацией о клиенте...
CREATE TYPE customer_detail_ot UNDER customer_ot
( first_name VARCHAR2(20)
, last_name  VARCHAR2(60)
, birth_date DATE
) FINAL;

-- Подтип с информацией об адресе...
CREATE TYPE address_detail_ot UNDER customer_ot
( address_id    NUMBER
, primary      VARCHAR2(1)
, street_address VARCHAR2(40)
, postal_code   VARCHAR2(10)
) FINAL;
```

Если вы никогда не работали с объектными типами, я рекомендую ознакомиться с материалом главы 26. В двух словах: поддержка заменяемости типов в Oracle означает, что я могу создавать строки как типа `customer_detail_ot`, так и `address_detail_ot`, и использовать их везде, где ожидается супертип `customer_ot supertype`. Таким образом, если я создаю конвейерную функцию для передачи коллекции супертипа, это означает, что я также могу передавать строки любого из подтипов. Это всего лишь один пример того, какие простые и элегантные решения можно создавать с использованием объектно-ориентированных иерархий типов.

Универсальная конвейерная функция

Взгляните, как выглядит тело конвейерной функции, а затем я объясню основные концепции:

```

/* Файл в Сети: multitype_setup.sql */
1  FUNCTION customer_transform_multi(
2      p_source      IN customer_staging_rct,
3      p_limit_size  IN PLS_INTEGER DEFAULT customer_pkg.c_default_limit
4  )
5      RETURN customer_ntt
6      PIPELINED
7      PARALLEL_ENABLE (PARTITION p_source BY HASH(customer_id))
8      ORDER p_source BY (customer_id, address_id) IS
9
10     aa_source      customer_staging_aat;
11     v_customer_id  customer_staging.customer_id%TYPE := -1;
12     /* Значение по умолчанию должно быть отлично от null */
13 BEGIN
14     LOOP
15         FETCH p_source BULK COLLECT INTO aa_source LIMIT p_limit_size;
16         EXIT WHEN aa_source.COUNT = 0;
17
18         FOR i IN 1 .. aa_source.COUNT LOOP
19
20             /* Передается только первый экземпляр описания клиента... */
21             IF aa_source(i).customer_id != v_customer_id THEN
22                 PIPE ROW ( customer_detail_ot( aa_source(i).customer_id,
23                                                 aa_source(i).first_name,
24                                                 aa_source(i).last_name,
25                                                 aa_source(i).birth_date ));
26             END IF;
27
28             PIPE ROW( address_detail_ot( aa_source(i).customer_id,
29                                         aa_source(i).address_id,
30                                         aa_source(i).primary,
31                                         aa_source(i).street_address,
32                                         aa_source(i).postal_code ));
33
34             /* Сохранение идентификатора для логики "прерывания"... */
35             v_customer_id := aa_source(i).customer_id;
36
37         END LOOP;
38     END LOOP;
39     CLOSE p_source;
40     RETURN;
41 END customer_transform_multi;

```

У этой функции включен параллельный режим, и она обрабатывает исходные данные в массивах для достижения максимальной производительности. Основные концепции, относящиеся к универсальности типов, описаны в следующей таблице.

| Строки | Описание |
|--------|---|
| 5 | Функция возвращает коллекцию с элементами супертипа, которые могут использоваться для передачи подтипов |
| 7–8 | В данных присутствуют зависимости, поэтому используется хеш-группировка с упорядоченной потоковой передачей. Записи каждого клиента должны обрабатываться вместе, потому что атрибуты клиента берутся только из первой записи |
| 21–26 | Если это первая запись конкретного клиента, передается строка типа CUSTOMER_DETAIL_OT. Для каждого клиента передается только одна запись с информацией о клиенте |
| 28–32 | Для каждой записи извлекается адресная информация и передается строка типа ADDRESS_DETAIL_OT |

Универсальная конвейерная функция в запросах

Итак, у меня имеется одна функция, способная генерировать строки двух разных типов и структур. Попробуем запросить несколько строк из этой функции в SQL*Plus:

```
/* Файл в Сети: multitype_query.sql */
SQL> SELECT *
  2 FROM TABLE(
  3     customer_pkg.customer_transform_multi(
  4     CURSOR( SELECT * FROM customer_staging ) ) ) nt
  5 WHERE ROWNUM <= 5;
```

CUSTOMER_ID

```
-----
1
1
1
1
2
```

Сюрприз — где данные? Хотя я использовал `SELECT *`, в результатах присутствует только столбец `CUSTOMER_ID`. Причина проста: согласно определению, моя функция возвращает коллекцию с элементами супертипа `customer_ot`, который имеет всего один атрибут. Без явного программирования диапазона подтипов, возвращаемых функцией, база данных не покажет дополнительные атрибуты. Более того, при попытке обратиться к любому из атрибутов подтипов в приведенном формате запроса база данных инициирует исключение `ORA-00904` (недействительный идентификатор).

К счастью, Oracle предоставляет два взаимозаменяемых способа обращения к экземплярам объектных типов: функцию `VALUE` и псевдостолбец `OBJECT_VALUE`. Посмотрим, как они работают:

```
/* Файл в Сети: multitype_query.sql */
SQL> SELECT VALUE(nt) AS object_instance -- также можно использовать nt.OBJECT_VALUE
  2 FROM TABLE(
  3     customer_pkg.customer_transform_multi(
  4     CURSOR( SELECT * FROM customer_staging ) ) ) nt
  5 WHERE ROWNUM <= 5;
```

OBJECT_INSTANCE(CUSTOMER_ID)

```
-----
CUSTOMER_DETAIL_OT(1, 'Abigail', 'Kessel', '31/03/1949')
ADDRESS_DETAIL_OT(1, 12135, 'N', '37 North Coshocton Street', '78247')
ADDRESS_DETAIL_OT(1, 12136, 'N', '47 East Sagadahoc Road', '90285')
ADDRESS_DETAIL_OT(1, 12156, 'Y', '7 South 3rd Circle', '30828')
CUSTOMER_DETAIL_OT(2, 'Anne', 'KOCH', '23/09/1949')
```

Уже лучше. Теперь данные выглядят так, как их возвращает конвейерная функция, и я выполняю с ними две операции. Сначала я определяю тип каждой записи конструкцией `IS OF`, а затем использую функцию `TREAT` для понижения каждой записи к ее фактическому подтипу (до этого база данных считала, что мои данные относятся к супертипу, и не позволит обратиться к их атрибутам).

Запрос выглядит примерно так:

```
/* Файл в Сети: multitype_query.sql */
SQL> SELECT CASE
  2     WHEN VALUE(nt) IS OF TYPE (customer_detail_ot)
  3     THEN 'C'
  4     ELSE 'A'
  5     END
  6     , TREAT(VALUE(nt) AS customer_detail_ot) AS cust_rec
  7     , TREAT(VALUE(nt) AS address_detail_ot) AS addr_rec
```

продолжение ➤

```

8 FROM TABLE(
9     customer_pkg.customer_transform_multi(
10         CURSOR( SELECT * FROM customer_staging ) ) ) nt
11 WHERE ROWNUM <= 5;

```

| RECORD_TYPE | CUST_REC | ADDR_REC |
|-------------|--|--|
| C | CUSTOMER_DETAIL_OT(1, 'Abigail', 'Kessel', '31/03/1949') | |
| A | | ADDRESS_DETAIL_OT(1, 12135, 'N', '37 North Coshocton Street', '78247') |
| A | | ADDRESS_DETAIL_OT(1, 12136, 'N', '47 East Sagadahoc Road', '90285') |
| A | | ADDRESS_DETAIL_OT(1, 12156, 'Y', '7 South 3rd Circle', '30828') |
| C | CUSTOMER_DETAIL_OT(2, 'Anne', 'KOCH', '23/09/1949') | |

Теперь данные имеют формат правильного подтипа, что позволяет мне обратиться к их атрибутам. Для этого предыдущий запрос упаковывается во встроенное представление, а в обращениях к атрибутам используется «точечная» запись:

```

/* Файл в Сети: multitype_query.sql */
SELECT ilv.record_type
,      NVL(ilv.cust_rec.customer_id,
          ilv.addr_rec.customer_id) AS customer_id
,      ilv.cust_rec.first_name      AS first_name
,      ilv.cust_rec.last_name       AS last_name
,      <snip>
,      ilv.addr_rec.postal_code      AS postal_code
FROM (
    SELECT CASE...
        <...>
    FROM TABLE(
        customer_pkg.customer_transform_multi(
            CURSOR( SELECT * FROM customer_staging ) ) ) nt
    ) ilv;

```

Загрузка нескольких таблиц из универсальной конвейерной функции

Я удалил несколько строк из предыдущего примера, но закономерность уже понятна. Теперь у меня имеются все элементы, необходимые для многотабличной вставки в таблицы клиентов и адресов. Код загрузки данных выглядит так:

```

/* Файл в Сети: multitype_setup.sql */
INSERT FIRST
    WHEN record_type = 'C'
    THEN
        INTO customers
        VALUES (customer_id, first_name, last_name, birth_date)
    WHEN record_type = 'A'
    THEN
        INTO addresses
        VALUES (address_id, customer_id, primary, street_address,
                postal_code)
SELECT ilv.record_type

```

```

,      NVL(ilv.cust_rec.customer_id,
          ilv.addr_rec.customer_id) AS customer_id
,      ilv.cust_rec.first_name      AS first_name
,      ilv.cust_rec.last_name       AS last_name
,      ilv.cust_rec.birth_date      AS birth_date
,      ilv.addr_rec.address_id      AS address_id
,      ilv.addr_rec.primary         AS primary
,      ilv.addr_rec.street_address  AS street_address
,      ilv.addr_rec.postal_code     AS postal_code
FROM (
  SELECT CASE
    WHEN VALUE(nt) IS OF TYPE (customer_detail_ot)
    THEN 'C'
    ELSE 'A'
  END
        AS record_type
,      TREAT(VALUE(nt) AS customer_detail_ot) AS cust_rec
,      TREAT(VALUE(nt) AS address_detail_ot)  AS addr_rec
FROM    TABLE(
        customer_pkg.customer_transform_multi(
          CURSOR( SELECT * FROM customer_staging ))) nt
) ilv;

```

Команда **INSERT FIRST** позволяет выполнить сложную загрузку данных, использующую объектно-ориентированные возможности. Скорее всего, этот способ подойдет и вам.

Альтернативное решение

Также возможен другой путь: создание одной «широкой» объектной записи с передачей одной строки для каждого набора адресов. Следующее определение типа поможет понять, о чем я говорю, но полный код приведен в файлах `multitype_setup.sql` на сайте книги:

```

/* Файл в Сети: multitype_setup.sql */
CREATE TYPE customer_address_ot AS OBJECT
( customer_id      NUMBER
, first_name      VARCHAR2(20)
, last_name       VARCHAR2(60)
, birth_date      DATE
, addr1_address_id NUMBER
, addr1_primary   VARCHAR2(1)
, addr1_street_address VARCHAR2(40)
, addr1_postal_code VARCHAR2(10)
, addr2_address_id NUMBER
, addr2_primary   VARCHAR2(1)
, addr2_street_address VARCHAR2(40)
, addr2_postal_code VARCHAR2(10)
, addr3_address_id NUMBER
, addr3_primary   VARCHAR2(1)
, addr3_street_address VARCHAR2(40)
, addr3_postal_code VARCHAR2(10)
, CONSTRUCTOR FUNCTION customer_address_ot
  RETURN SELF AS RESULT
);

```

Как видите, каждый из трех экземпляров адреса «денормализуется» на соответствующие атрибуты. Каждая строка, переданная из функции, трансформируется в четыре строки условной командой **INSERT ALL**. Синтаксис **INSERT** проще (а в этом конкретном примере быстрее) метода с заменяемыми типами. Выбор метода зависит от конкретных обстоятельств; однако следует учесть, что при возрастании количества атрибутов производительность денормализованного метода может ухудшиться. Впрочем, я успешно использовал этот метод для оптимизации загрузки данных, вставляющей до 9 записей в 4 таблицы для каждой финансовой операции в торговом приложении.



При использовании строк со многими столбцами (по аналогии с приведенным денормализованным примером) следует ожидать ухудшения производительности реализации с конвейерной функцией. Например, я проводил тестирование последовательной массовой конвейерной загрузки 50 000 строк по сравнению с построчной вставкой, использующей большое количество столбцов по 10 байт. В Oracle9i второе решение стало превосходить конвейерное уже на 50 столбцах. К счастью, во всех основных версиях Oracle Database 10g и Oracle Database 11g этот порог находится где-то в диапазоне от 100 до 150 столбцов.

В завершение о конвейерных функциях

При обсуждении конвейерных функций я привел несколько сценариев, в которых такие функции (последовательные или параллельные) могут повысить эффективность загрузки и извлечения данных. Некоторые из описанных методов пригодятся как средство оптимизации. Однако я не рекомендую переводить всю кодовую базу на конвейерные функции! Это специфический инструмент, который находит применение лишь в ограниченном подмножестве задач обработки данных. Но если вам понадобится реализовать сложные преобразования, слишком громоздкие для представления в SQL (обычно это аналитические функции, выражения CASE, подзапросы и даже пресловутая секция MODEL), инкапсуляция их в конвейерных функциях способна обеспечить существенный прирост эффективности.

Специализированные приемы оптимизации

Всегда используйте FORALL и BULK COLLECT для любых нетривиальных многострочных операций SQL (то есть операций, в которых задействовано более нескольких десятков строк). *Всегда* ищите возможность организовать кэширование данных. А для многих задач обработки данных следует активно рассмотреть использование конвейерных функций. Иначе говоря, некоторые приемы оптимизации настолько эффективны, что должны использоваться при любой возможности.

Другие приемы оптимизации работают только в специальных ситуациях. Например, замена типа данных INTEGER типом PLS_INTEGER принесет сколько-нибудь заметную пользу только в программе, интенсивно обрабатывающей числовые данные.

Именно этой теме и посвящен настоящий раздел: в нем рассматриваются функции PL/SQL, способные значительно повысить производительность, но только в особых ситуациях. В общем случае вам не стоит изначально беспокоиться о применении всех этих возможностей. Направьте усилия на разработку удобочитаемого, легкого в сопровождении кода, а если впоследствии в программе обнаружатся узкие места, посмотрите, не помогут ли вам какие-либо из описанных приемов.

Метод передачи параметров NOCOPY

Режим передачи параметров NOCOPY требует, чтобы ядро PL/SQL передавало аргументы IN OUT по ссылке, а не по значению. Передача по ссылке повышает производительность программ, потому что она предотвращает копирование аргументов в программный модуль. При передаче больших, сложных структур (коллекций, записей или объектов) копирование данных может быть сопряжено с заметными затратами.

Чтобы понять смысл режима NOCOPY и его возможное влияние на производительность, необходимо ознакомиться с тем, как PL/SQL передаются параметры. Существует два способа передачи:

- **По ссылке.** С соответствующим формальным параметром связывается указатель, а не фактическое значение. После этого и формальный и фактический параметры ссылаются на одну ячейку памяти, содержащую значение параметра.
- **По значению.** Значение фактического параметра копируется в соответствующий формальный параметр. Если впоследствии программа завершается без инициирования исключений, значение формального параметра присваивается фактическому. В случае же возникновения ошибки измененное значение обратно не передается.

Без использования `NOCOPY` передача параметров в PL/SQL выполняется по следующим правилам.

| Режим | Передача (по умолчанию) |
|--------|-------------------------|
| IN | По ссылке |
| OUT | По значению |
| IN OUT | По значению |

Из этих определений и правил следует, что передача большой структуры в режиме `OUT` или `IN OUT` производится по значению, а копирование может привести к потере производительности и лишним затратам памяти. Для предотвращения всех этих неприятностей и служит секция `NOCOPY`. Она включается в объявление параметра следующим образом:

имя_параметра [IN | IN OUT | OUT | IN OUT `NOCOPY` | OUT `NOCOPY`] *тип_данных_параметра*

Секция `NOCOPY` используется только совместно с режимами `OUT` и `IN OUT`. Пример списка параметров с секцией `NOCOPY`:

```
PROCEDURE analyze_results (
    date_in IN DATE,
    values IN OUT NOCOPY numbers_varray,
    validity_flags IN OUT NOCOPY validity_rectype
);
```

Применяя секцию `NOCOPY`, следует учитывать два важных обстоятельства:

- Каждый раз при вызове подпрограммы с выходным параметром, объявленным с `NOCOPY`, значение фактического параметра, соответствующего выходному, устанавливается в `NULL`.
- Секция `NOCOPY` — это рекомендация, а не команда, поэтому не исключено, что компилятор самостоятельно примет решение о невозможности выполнения запроса. Ограничения на использование `NOCOPY` перечислены в следующем разделе.

Ограничения на использование `NOCOPY`

Иногда компилятор PL/SQL игнорирует секцию `NOCOPY` и использует метод передачи по значению, который задан для параметров с режимами `OUT` и `IN OUT` по умолчанию. Это может произойти в следующих ситуациях:

- **Фактический параметр является элементом ассоциативного массива.** Секция `NOCOPY` может применяться ко всей коллекции, но не к ее отдельным элементам. Данное ограничение легко обойти: скопируйте структуру в переменную, скаляр или запись, а затем передайте ее как параметр, объявление которого содержит секцию `NOCOPY`.
- **Некоторые ограничения на фактические параметры.** Под действием некоторых ограничений секция `NOCOPY` игнорируется. Например, к ним относится указание количества знаков в дробной части для числовых значений или ограничение `NOT NULL`. На строковые переменные, объявленные с ограничением длины, это не распространяется.

- **Фактические и формальные параметры представляют собой записи.** Одна или обе записи объявлены с использованием атрибута %ROWTYPE или %TYPE, а ограничения на соответствующие поля этих двух записей разные.
- **Передача фактического параметра требует неявного преобразования типов данных.** Выход из этой ситуации может быть следующим: поскольку всегда лучше использовать явное преобразование типов данных, выполните его, а затем передавайте преобразованные значения как параметры, в объявлении которых имеется секция NOCOPY.
- **Обращение к подпрограмме осуществляется путем внешнего или удаленного вызова процедуры.** В таких ситуациях PL/SQL всегда будет передавать фактические параметры по значению.

Эффективность NOCOPY

Итак, насколько же NOCOPY ускорит вашу программу? Чтобы ответить на этот вопрос, я создал пакет с двумя процедурами:

```
/* Файл в Сети: nocopy_performance.tst */
PACKAGE nocopy_test
IS
  TYPE numbers_t IS TABLE OF NUMBER;
  PROCEDURE pass_by_value (numbers_inout IN OUT numbers_t);
  PROCEDURE pass_by_ref (numbers_inout IN OUT NOCOPY numbers_t);
END nocopy_test;
```

Каждая процедура удваивает значение элемента вложенной таблицы:

```
PROCEDURE pass_by_value (numbers_inout IN OUT numbers_t)
IS
BEGIN
  FOR indx IN 1 .. numbers_inout.COUNT
  LOOP
    numbers_inout (indx) := numbers_inout (indx) * 2;
  END LOOP;
END;
```

Затем для каждой процедуры:

- Вложенная таблица была заполнена 100 000 строк данных.
- Процедура была вызвана 1000 раз.

В Oracle Database 10g были получены следующие результаты:

```
By value (without NOCOPY) - Elapsed CPU : 20.49 seconds.
By reference (with NOCOPY) - Elapsed CPU : 12.32 seconds.
```

Однако в Oracle Database 11g результаты были такими:

```
By value (without NOCOPY) - Elapsed CPU : 13.12 seconds.
By reference (with NOCOPY) - Elapsed CPU : 12.82 seconds.
```

Я провел аналогичные тесты с коллекциями строк — с аналогичными результатами.

После проведения многократных тестов я делаю вывод, что до Oracle Database 11g наблюдался значительный прирост производительности, а в Oracle Database 11g этот прирост существенно сократился. Как я полагаю, это объясняется общей оптимизацией ядра PL/SQL в новой версии.

Недостатки NOCOPY

В зависимости от приложения конструкция NOCOPY может повысить производительность программ с параметрами IN OUT и OUT. Однако за потенциальный прирост приходится расплачиваться: если программа завершается с необработанным исключением, вы не можете доверять значениям в фактическом параметре NOCOPY.

Что значит «доверять»? Давайте разберемся, что происходит в PL/SQL с параметрами при завершении программы с необработанным исключением. Предположим, я передаю своей процедуре `calculate_totals` запись `IN OUT`. Исполнительное ядро PL/SQL сначала создает копию этой записи, а затем во время выполнения программы вносит изменения в копию. Фактический параметр не изменяется до того момента, когда процедура `calculate_totals` успешно завершится (без передачи исключения). В этот момент локальная копия переносится в фактический параметр, а программа, вызвавшая `calculate_totals`, может обратиться к измененным данным. Но если процедура `calculate_totals` завершится с необработанным исключением, вызывающая программа может быть уверена в том, что значение фактического параметра осталось неизменным.

С рекомендацией `NOCOPY` эта уверенность исчезает. При передаче параметра по ссылке (эффект `NOCOPY`) любые изменения, вносимые в формальный параметр, также немедленно вносятся в фактический параметр. Предположим, программа `calculate_totals` читает коллекцию из 10 000 строк и вносит изменения в каждую строку. Если ошибка возникла в строке 5000 и была передана из `calculate_totals` необработанной, коллекция (фактический параметр) будет изменена только наполовину.

Файл `nocopy.tst` на сайте книги демонстрирует проблемы с использованием `NOCOPY`. Запустите сценарий и разберитесь во всех тонкостях этого режима передачи параметров, прежде чем использовать его в своих приложениях.

В целом, будьте внимательны при использовании рекомендации `NOCOPY`. Используйте ее только тогда, когда вы твердо уверены в наличии проблем, связанных с передачей параметров; будьте готовы к возможным последствиям при инициировании исключений.



Руководитель разработки PL/SQL Брин Луэллин придерживается иных взглядов на `NOCOPY`: он рекомендует широко применять эту возможность в программах. Брин полагает, что побочные эффекты частично измененных структур данных не должны представлять большой опасности, потому что эта ситуация возникает только при появлении непредвиденных ошибок. В таких случаях почти всегда следует прервать работу приложения, сохранить информацию об ошибке и передать исключение во внешний блок. Тот факт, что коллекция находится в неопределенном состоянии, в этот момент уже не важен.

Выбор типа данных

При выполнении относительно небольшого количества операций не так уж важно, будет ли PL/SQL выполнять неявные преобразования или использовать относительно медленную реализацию. С другой стороны, для алгоритмов, сопряженных с большими объемами вычислений, следующие рекомендации могут обеспечить заметный выигрыш.

Избегайте неявных преобразований

PL/SQL, как и SQL, во многих ситуациях выполняет неявные преобразования. Так, в следующем блоке PL/SQL сначала преобразует целочисленное значение 1 в вещественный формат 1.0, прежде чем суммировать его с другим числом и присвоить результат переменной типа `NUMBER`:

```
DECLARE
  l_number NUMBER := 2.0;
BEGIN
  l_number := l_number + 1;
END;
```

Многие разработчики знают, что выполнение неявных преобразований в командах SQL может привести к снижению производительности. Неявные преобразования в PL/SQL тоже могут повлиять на производительность, хотя и не так серьезно, как преобразования в SQL.

Чтобы проверить влияние неявных преобразований на производительность в вашей среде, запустите сценарий `test_implicit_conversion.sql`.

Используйте тип `PLS_INTEGER` для интенсивных целочисленных вычислений

Целочисленная переменная, объявленная с типом `PLS_INTEGER`, расходует меньше памяти по сравнению с `INTEGER` и выполняет свою работу более эффективно за счет использования аппаратной поддержки вычислений. В программе с интенсивной обработкой числовых данных даже простое изменение типа может заметно повлиять на производительность. Более подробная информация о разных целочисленных типах приведена в разделе «Тип `PLS_INTEGER`» главы 9.

Используйте тип `BINARY_FLOAT` или `BINARY_DOUBLE` для интенсивных вещественных вычислений

В Oracle10g появились два новых вещественных типа, `BINARY_FLOAT` и `BINARY_DOUBLE`. Они соответствуют стандарту IEEE 754 и используют машинную реализацию математических операций, что делает их более эффективными по сравнению с `NUMBER` или `INTEGER`. За дополнительной информацией обращайтесь к разделу «Типы `BINARY_FLOAT` и `BINARY_DOUBLE`» главы 9.

Оптимизация вызовов функций в SQL (версия 12.1 и выше)

Oracle Database 12c предоставляет два важных усовершенствования, повышающих эффективность вызова функций PL/SQL в командах SQL:

- Секция `WITH FUNCTION`.
- Директива `UDF`.

Синтаксис `WITH FUNCTION` более подробно рассматривается в главе 17.

Директива `UDF` предоставляет гораздо более простое средство повышения эффективности вызовов функций из SQL. Чтобы воспользоваться ею, достаточно добавить следующую строку в раздел объявлений функции:

```
PRAGMA UDF;
```

Эта директива сообщает Oracle: «Эта функция будет в основном вызываться из SQL, а не из блоков PL/SQL». Oracle использует эту информацию для сокращения затрат на переключения контекста с SQL на PL/SQL при выполнении этой функции.

В результате функция значительно быстрее выполняется из SQL (руководитель группы разработки PL/SQL утверждает, что возможно четырехкратное повышение производительности), но при выполнении из блока PL/SQL она будет работать чуть медленнее (!).

Сценарий в файле `12c_udf.sql` демонстрирует использование этой возможности. Он сравнивает производительность функций с включенным режимом `UDF` и без него. Проверьте сами и посмотрите, какой выигрыш может принести использование этой очень простой директивы!

Подведем итог: сначала попробуйте использовать директиву `UDF`. Если она не обеспечит заметного прироста скорости, тогда пробуйте `WITH FUNCTION`.

Общие замечания о производительности

В этой главе были описаны разные способы повышения производительности программ PL/SQL. Почти все они сопряжены с компромиссами: повышение производительности достигается за счет дополнительных затрат памяти, усложнения кода и затрат на сопровождение и т. д. Следующие рекомендации помогут вам оптимизировать код так, чтобы это принесло максимальную пользу как пользователям, так и группе разработки:

- Проверьте эффективность всех команд SQL. Оптимизация кода PL/SQL просто не компенсирует «балласта» в виде ненужных просмотров целых таблиц. Если код SQL работает медленно, в PL/SQL эту проблему не решить.
- Убедитесь в том, что оптимизация PL/SQL выполняется как минимум на уровне 2. Этот уровень используется по умолчанию, но разработчики нередко «экспериментируют» с параметрами компилятора, и в результате у них получается код, не полностью оптимизируемый компилятором. Директива условной компиляции `$ERROR` позволяет принудительно задать этот уровень (см. главу 20).
- Используйте конструкции `BULK COLLECT` и `FORALL` при каждой возможности. Скажем, если в вашей программе выполняются запросы или команды DML, работающие на уровне отдельных строк данных, вам стоит переработать код для работы с SQL на уровне коллекций. Переработка курсорных циклов `FOR` не столь критична, но конструкции `OPEN...LOOP...CLOSE` всегда выбирают записи по одной и их следует переработать.
- Уделяйте особое внимание статическим наборам данных. Обнаружив такой набор, выберите оптимальный метод кэширования для предотвращения повторяющейся, высокозатратной выборки данных. Даже если вы еще не используете Oracle Database 11g и выше, начните инкапсулировать свои запросы в интерфейсах функций. Это позволит вам легко и быстро применить кэширование результатов функций при переходе на новую версию Oracle.
- Ваш код не обязан быть «самым быстрым из всех возможных» — вполне достаточно, если он будет «в меру быстрым». Иначе говоря, не пытайтесь фанатично оптимизировать каждую строку кода. Удобство чтения и сопровождения важнее «молниеносной быстроты». Сначала добейтесь того, чтобы ваш код соответствовал требованиям пользователя, а затем протестируйте его и выявите возможные узкие места. Избавьтесь от них, применяя специализированные приемы оптимизации.
- Убедитесь в том, что администратор базы данных хорошо знает параметры компиляции во внутренний код (особенно в Oracle 11g и выше). С этими параметрами Oracle автоматически преобразует код PL/SQL в команды машинного кода.

22 Ввод/вывод в PL/SQL

Большинство программ PL/SQL работает только с базой данных Oracle через SQL. Однако время от времени возникает необходимость в передаче информации из PL/SQL во внешнюю среду или чтении информации из внешнего источника (экран, файл и т. д.) в PL/SQL. В этой главе рассматриваются самые распространенные механизмы ввода/вывода в PL/SQL, в числе которых следующие встроенные пакеты:

- DBMS_OUTPUT — вывод информации на экран;
- UTL_FILE — чтение и запись файлов операционной системы;
- UTL_MAIL и UTL_SMTP — отправка электронной почты из PL/SQL;
- UTL_HTTP — получение данных с веб-страниц.

Полное описание этих встроенных пакетов выходит за рамки книги. В этой главе будут представлены примеры их применения для реализации часто встречающихся требований. За более полной информацией обращайтесь к документации Oracle. Полезная информация о многих пакетах также приведена в книге *Oracle Built-in Packages* (издательство O'Reilly); некоторые главы из которой доступны на сайте данной книги.

Вывод информации

Пакет DBMS_OUTPUT предоставляет средства вывода информации из программ в буфер. Далее содержимое буфера читается и обрабатывается другой программой PL/SQL или управляющей средой. Пакет DBMS_OUTPUT чаще всего используется для простого вывода информации на экран.

Каждому пользовательскому сеансу выделяется буфер DBMS_OUTPUT заранее определенного размера. В Oracle10g Release 2 действовало ограничение в один миллион байтов. Когда буфер будет заполнен, его необходимо очистить перед дальнейшим использованием; это можно сделать на программном уровне, но чаще для очистки и вывода содержимого буфера используется управляющая среда (например, SQL*Plus). Это происходит только после завершения внешнего блока PL/SQL; пакет DBMS_OUTPUT не может использоваться для передачи сообщений из программы в реальном времени.

Запись информации в буфер выполняется при помощи программ DBMS_OUTPUT.PUT и DBMS_OUTPUT.PUT_LINE, а чтение на программном уровне — при помощи программ DBMS_OUTPUT.GET_LINE или DBMS_OUTPUT.GET_LINES.

Включение DBMS_OUTPUT

По умолчанию пакет DBMS_OUTPUT отключен, поэтому вызовы программ PUT_LINE и PUT игнорируются, а буфер остается пустым. Обычно включение DBMS_OUTPUT осуществляется специальной командой в управляющей среде. Например, в программе SQL*Plus выполняется следующая команда:

```
SET SERVEROUTPUT ON SIZE UNLIMITED
```

Кроме включения вывода на консоль, эта команда передает серверу базы данных следующую команду:

```
BEGIN DBMS_OUTPUT.ENABLE (buffer_size => NULL); END;
```

(Параметр buffer_size со значением NULL означает неограниченный буфер; также может передаваться конкретное значение размера в байтах.) SQL*Plus поддерживает ряд дополнительных параметров команды SERVEROUTPUT; за дополнительной информацией обращайтесь к документации своей версии.

Среды разработки (например, Oracle SQL Developer или Quest Toad) обычно выводят результаты DBMS_OUTPUT в специально назначенной части экрана — конечно, для этого вывод должен быть предварительно включен.

Запись в буфер

Запись информации в буфер осуществляется двумя встроенными процедурами: PUT_LINE присоединяет маркер новой строки после текста, а PUT помещает текст в буфер без маркера новой строки. Информация, выведенная PUT, останется в буфере даже при завершении вызова. В этом случае следует вывести ее с очисткой буфера вызовом DBMS_OUTPUT.NEW_LINE.

Если Oracle знает, как неявно преобразовать ваши данные в строку VARCHAR2, то эти данные можно передать при вызове программ PUT и PUT_LINE. Примеры:

```
BEGIN
  DBMS_OUTPUT.put_line ('Steven');
  DBMS_OUTPUT.put_line (100);
  DBMS_OUTPUT.put_line (SYSDATE);
END;
/
```

К сожалению, DBMS_OUTPUT не умеет выполнять такое преобразование для многих распространенных типов PL/SQL, прежде всего для BOOLEAN. В таких случаях стоит написать небольшую утилиту, упрощающую вывод логических значений:

```
/* Файл в Сети: bp1.sp */
PROCEDURE bp1 (boolean_in IN BOOLEAN)
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    CASE boolean_in
      WHEN TRUE THEN 'TRUE'
      WHEN FALSE THEN 'FALSE'
      ELSE 'NULL'
    END
  );
END bp1;
/
```

Максимальная длина строки, которая может передаваться за один вызов DBMS_OUTPUT.PUT_LINE, в последних версиях Oracle составляет 32 767 байт. В Oracle10g Release 1 и ранее действовало ограничение в 255 байт. В любой версии при передаче строки с длиной

более допустимого максимума происходит исключение (VALUE_ERROR или ORU-10028). Чтобы избавиться от этой проблемы, используйте инкапсуляцию DBMS_OUTPUT.PUT_LINE, которая автоматически разрывает длинные строки. В следующих файлах, доступных на сайте книги, представлены вариации на эту тему:

- pl.sp — автономная процедура позволяет указать длину разрыва строки.
- p.pks/pkb — пакет p представляет собой масштабную инкапсуляцию DBMS_OUTPUT.PUT_LINE с множеством разных перегрузок (например, вызовом процедуры p.l можно вывести документ XML или файл операционной системы), а также с возможностью разрыва длинного текста.

Чтение содержимого буфера

Типичный случай использования DBMS_OUTPUT весьма тривиален: разработчик вызывает DBMS_OUTPUT.PUT_LINE и просматривает результаты на экране. Клиентская среда (например, SQL*Plus) «за кулисами» вызывает соответствующие программы пакета DBMS_OUTPUT для чтения и вывода содержимого буфера.

Для получения содержимого буфера DBMS_OUTPUT используются процедуры GET_LINE и/или GET_LINES.

Процедура GET_LINE читает одну строку из буфера по принципу «первым пришел, первым вышел» и в случае успешного завершения возвращает код 0. Пример использования программы для чтения следующей строки из буфера в локальную переменную PL/SQL:

```
FUNCTION next_line RETURN VARCHAR2
IS
    return_value VARCHAR2(32767);
    status INTEGER;
BEGIN
    DBMS_OUTPUT.GET_LINE (return_value, status);
    IF status = 0
    THEN
        RETURN return_value;
    ELSE
        RETURN NULL;
    END IF;
END;
```

Процедура GET_LINES читает несколько строк из буфера за один вызов. Данные загружаются в коллекцию строк PL/SQL (максимальная длина от 255 до 32 767 в зависимости от версии Oracle). Вы указываете количество читаемых строк, а процедура возвращает их. Следующая программа записывает содержимое буфера DBMS_OUTPUT в журнальную таблицу базы данных:

```
/* Файл в Сети: move_buffer_to_log.sp */
PROCEDURE move_buffer_to_log
IS
    l_buffer      DBMS_OUTPUT.chararr;
    l_num_lines   PLS_INTEGER;
BEGIN
    LOOP
        l_num_lines := 100;
        DBMS_OUTPUT.get_lines (l_buffer, l_num_lines);
        EXIT WHEN l_buffer.COUNT = 0;
        FORALL indx IN l_buffer.FIRST .. l_buffer.LAST
            INSERT INTO logtab (text) VALUES (l_buffer (indx));
        END LOOP;
END;
```

Чтение и запись файлов

Пакет `UTL_FILE` позволяет программам PL/SQL как читать, так и записывать данные в любой файл операционной системы, доступный для сервера, на котором работает экземпляр базы данных. Данные из файла можно загрузить непосредственно в таблицу базы данных, в полной мере используя всю мощь и гибкость программирования PL/SQL.

Также разработчик может генерировать отчеты прямо из PL/SQL, не беспокоясь об ограничениях на максимальный размер буфера `DBMS_OUTPUT`, существовавших до выхода Oracle Database 10g Release 2.

Идея записи произвольных файлов выглядит довольно рискованно — злонамеренный или беспечный программист теоретически может воспользоваться `UTL_FILE` для записи файлов баз данных табличного пространства, управляющих файлов и т. д. Oracle позволяет администратору установить ограничения на чтение и запись файлов одним из двух способов:

- Чтение и запись файлов средствами `UTL_FILE` производится в каталогах, заданных параметром `UTL_FILE_DIR` в инициализационном файле базы данных.
- `UTL_FILE` также читает и записывает файлы в каталогах, определяемых объектами каталогов базы данных.

Сначала вы узнаете, как использовать эти два механизма, а затем мы рассмотрим специфические возможности пакета `UTL_FILE`. Многие программы `UTL_FILE` инкапсулируются в удобном пакете, доступном в файле `fileIO.pkg` на сайте книги.

Параметр `UTL_FILE_DIR`

Хотя использование `UTL_FILE_DIR` официально не признано устаревшим, этот подход редко используется в новейших версиях базы данных Oracle. Объекты каталогов гораздо удобнее и обладают большей гибкостью. Если у вас есть выбор, не используйте `UTL_FILE_DIR`; просто пропустите этот раздел и переходите к разделу «Работа с каталогами в Oracle».

Вызывая `FOPEN` для открытия файла, необходимо указать как местонахождение, так и имя файла. Местонахождение файла проверяется по списку доступных каталогов, которые задаются в файле инициализации базы данных строками вида:

```
UTL_FILE_DIR = каталог
```

Включите параметр `UTL_FILE_DIR` для каждого каталога, который вы хотите сделать доступным для операций `UTL_FILE`. Например, следующие записи разрешают доступ к четырем разным каталогам в файловых системах семейства Unix/Linux:

```
UTL_FILE_DIR = /tmp
UTL_FILE_DIR = /ora_apps/hr/time_reporting
UTL_FILE_DIR = /ora_apps/hr/time_reporting/log
UTL_FILE_DIR = /users/test_area
```

Чтобы обойти средства безопасности сервера и разрешить чтение/запись во все каталоги, можно воспользоваться специальным синтаксисом:

```
UTL_FILE_DIR = *
```

Никогда не используйте этот механизм в среде реальной эксплуатации. Конечно, в среде разработки этот синтаксис упрощает использование `UTL_FILE` и тестирование кода. Однако при переводе приложения в рабочую среду доступ следует ограничить несколькими конкретными каталогами.

Настройка каталогов

Несколько замечаний по поводу настройки доступных каталогов для UTL_FILE и работы с ними:

- Доступ не распространяется на подкаталоги. Допустим, файл инициализации содержит следующие строки:

```
UTL_FILE_DIR = c:\group\dev1
UTL_FILE_DIR = c:\group\prod\oe
UTL_FILE_DIR = c:\group\prod\ar
```

Вы не сможете открыть файл в подкаталоге c:\group\prod\oe\reports.

- Не используйте следующую запись в системах Unix и Linux:

```
UTL_FILE_DIR = .
```

Она выполняет чтение/запись в текущем каталоге операционной системы.

- Не заключайте имена каталогов в одиночные или двойные кавычки.
- В среде Unix/Linux владельцем файла, созданного FOPEN, является теневой процесс, запустивший экземпляр Oracle (как правило, «oracle»). При попытке обратиться к этим файлам или изменить их за пределами UTL_FILE необходимо обладать соответствующими привилегиями (или войти в систему как «oracle»).
- Не завершайте имя каталога ограничителем (например, косой чертой в Unix/Linux). Следующая спецификация создаст проблемы при попытке чтения или записи в каталог:

```
UTL_FILE_DIR = /tmp/orafiles/
```

Определение местонахождения файла при открытии

Местонахождение файла определяется строкой, специфической для операционной системы, которая задает каталог или область для открытия файла. При передаче местонахождения в вызове UTL_FILE.FOPEN спецификация местонахождения указывается в том виде, в каком она задана в файле инициализации базы данных. И помните, что в операционных системах, учитывающих регистр символов, регистр спецификации в файле инициализации должен совпадать с регистром, использованным в вызове UTL_FILE.FOPEN.

Несколько примеров:

- Windows:

```
file_id := UTL_FILE.FOPEN ('k:\common\debug', 'trace.lis', 'R');
```

- Unix/Linux:

```
file_id := UTL_FILE.FOPEN ('/usr/od2000/admin', 'trace.lis', 'W');
```

Местонахождение должно задаваться явно, как полный путь к файлу. При этом не допускается использование параметров, относящихся к специфике операционной системы (например, переменных окружения в Unix/Linux).

Работа с каталогами в Oracle

До выхода версии Oracle9i Release 2 при открытии файла приходилось указывать его местонахождение. Однако жесткое кодирование данных всегда нежелательно — а если данные будут перемещены в другое место? Сколько программ придется изменять, чтобы они искали свои данные в правильном месте? И сколько раз придется вносить такие изменения?

Лучше объявить переменную или константу, которой будет присваиваться значение, определяющее местонахождение данных. Если сделать это в пакете, к константе можно

будет обращаться из любой программы в схеме, обладающей привилегией EXECUTE для этого пакета. Пример:

```
PACKAGE accts_pkg
IS
    c_data_location
        CONSTANT VARCHAR2(30) := '/accts/data';
    ...
END accts_pkg;

DECLARE
    file_id    UTL_FILE.file_type;
BEGIN
    file_id := UTL_FILE.fopen (accts_pkg.c_data_location, 'trans.dat', 'R');
END;
```

Неплохо, но еще лучше — использовать объект уровня схемы, который может определяться в базе данных. Для создания каталога администратор базы данных должен предоставить вам привилегию CREATE ANY DIRECTORY, после чего вы получаете возможность создания новых каталогов инструкциями следующего вида:

```
CREATE OR REPLACE DIRECTORY development_dir AS '/dev/source';
```

```
CREATE OR REPLACE DIRECTORY test_dir AS '/test/source';
```

Несколько важных обстоятельств, относящихся к созданию каталогов в UTL_FILE:

- Oracle не проверяет заданное вами имя каталога. Строка просто связывается с именованным объектом базы данных.
- Имя каталога, указанное, допустим, при вызове UTL_FILE.FOPEN, интерпретируется не как имя объекта Oracle, а как строка с учетом регистра символов. Иначе говоря, если имя не задается как строка с символами в верхнем регистре, операция завершится неудачей. Например, такое решение работает:

```
handle := UTL_FILE.FOPEN(
    location => 'TEST_DIR', filename => 'myfile.txt', open_mode =>
    'r');
```

... а такое — нет:

```
handle := UTL_FILE.FOPEN(
    location => test_dir, filename => 'myfile.txt', open_mode => 'r');
```

- После того как каталог будет создан, вы можете предоставить конкретным пользователям разрешения на работу с ним:

```
GRANT READ ON DIRECTORY development_dir TO senior_developer;
```

- Наконец, информацию о каталогах, доступных для текущей схемы, можно получить из представления ALL_DIRECTORIES. Данные представления также могут использоваться для создания полезных вспомогательных программ. В следующем примере выводится список всех каталогов, определенных в базе данных:

```
/* Файл в Сети: fileIO.pkg */
PROCEDURE fileIO.gen_utl_file_dir_entries
IS
BEGIN
    FOR rec IN (SELECT * FROM all_directories)
    LOOP
        DBMS_OUTPUT.PUT_LINE ('UTL_FILE_DIR = ' || rec.directory_path);
    END LOOP;
END gen_utl_file_dir_entries;
```

Одно из преимуществ подобных вспомогательных программ заключается в том, что они позволяют легко и нетривиально обрабатывать «ошибки форматирования» (например, если имя каталога не задается в верхнем регистре).

Открытие файлов

Прежде чем читать или записывать данные, файл необходимо предварительно открыть. Функция `UTL_FILE.FOPEN` открывает заданный файл и возвращает дескриптор для выполнения дальнейших операций с файлом. Заголовок функции выглядит так:

```
FUNCTION UTL_FILE.FOPEN (
    location IN VARCHAR2
    , filename IN VARCHAR2
    , open_mode IN VARCHAR2
    , max_linesize IN BINARY_INTEGER DEFAULT NULL)
RETURN UTL_FILE.file_type;
```

Здесь *location* — путь к файлу (каталог из `UTL_FILE_DIR` или объект каталога в базе данных); *filename* — имя файла; *open_mode* — режим открытия (см. ниже); *max_linesize* — максимальное количество символов в строке с учетом символа новой строки (от 1 до 32 767); `UTL_FILE.file_type` — запись со всей информацией, необходимой `UTL_FILE` для работы с файлом.

Файл может открываться в одном из трех режимов:

- **R** — файл открывается только для чтения. В этом режиме для чтения данных из файла используется процедура `GET_LINE` из пакета `UTL_FILE`.
- **W** — файл открывается для чтения и записи в режиме замены. Все существующие строки удаляются из файла. В этом режиме для модификации файла могут использоваться любые из следующих программ `UTL_FILE`: `PUT`, `PUT_LINE`, `NEW_LINE`, `PUTF` и `FFLUSH`.
- **A** — файл открывается для чтения и записи в режиме присоединения. Все существующие строки остаются в файле, а новые строки присоединяются за последней строкой. Для модификации файла могут использоваться любые из следующих программ `UTL_FILE`: `PUT`, `PUT_LINE`, `NEW_LINE`, `PUTF` и `FFLUSH`.

При открытии файлов необходимо учитывать следующие обстоятельства:

- Путь, объединенный с именем файла, должен представлять действительное имя файла в вашей операционной системе.
- Путь к файлу должен быть доступным и уже существующим; функция `FOPEN` не создает каталог или подкаталог для записи нового файла.
- Файл, открываемый для чтения, должен уже существовать. Если файл открывается для записи, он либо создается (если файл не существует), либо из него удаляется все содержимое (если файл существует).
- Файл, открываемый для присоединения, уже должен существовать. В противном случае `UTL_FILE` инициирует исключение `INVALID_OPERATION`.

Следующий пример демонстрирует объявление файлового дескриптора с последующим открытием файла только для чтения:

```
DECLARE
    config_file UTL_FILE.FILE_TYPE;
BEGIN
    config_file := UTL_FILE.FOPEN ('/maint/admin', 'config.txt', 'R');
```

Обратите внимание: при открытии файла максимальная длина строки не задается. Этот параметр не является обязательным. Если он не указан, то максимальная длина строки, которая читается или записывается в файл, равна приблизительно 1024. С учетом этого ограничения можно передать аргумент `max_linesize`, как это сделано в следующем примере:

```
DECLARE
    config_file UTL_FILE.FILE_TYPE;
BEGIN
    config_file := UTL_FILE.FOPEN (
        '/maint/admin', 'config.txt', 'R', max_linesize => 32767);
```




Файлы в многобайтовых кодировках следует открывать функцией `FOPEN_NCHAR`. В этом случае Oracle рекомендует ограничить `max_linesize` значением 6400.

Проверка открытия файла

Функция `IS_OPEN` возвращает `TRUE`, если заданный дескриптор указывает на уже открытый файл. В противном случае возвращается `FALSE`. Заголовок функции выглядит так:

```
FUNCTION UTL_FILE.IS_OPEN (file IN UTL_FILE.FILE_TYPE) RETURN BOOLEAN;
```

Здесь *file* — проверяемый файл.

Важно понимать, что означает такая проверка в контексте `UTL_FILE`. Функция `IS_OPEN` не выполняет никаких проверок на уровне операционной системы, она всего лишь убеждается в том, что поле `id` записи файлового дескриптора отлично от `NULL`. Если вы не модифицировали эти записи и их содержимое, поле `id` имеет отличное от `NULL` значение только при вызове `FOPEN`. При вызове `FCLOSE` в него снова записывается значение `NULL`.

Заккрытие файла

Для закрытия файла или всех открытых файлов в сеансе следует использовать процедуры `UTL_FILE.FCLOSE` и `UTL_FILE.FCLOSE_ALL` соответственно. Заголовок процедуры выглядит так:

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT UTL_FILE.FILE_TYPE);
```

Здесь *file* — имя закрываемого файла. Обратите внимание на то, что аргумент `UTL_FILE.FCLOSE` передается в режиме `IN OUT`, потому что процедура после закрытия файла записывает в поле `id` записи значение `NULL`.

Если при закрытии файла в буфере остались незаписанные данные, `UTL_FILE` инициирует исключение `WRITE_ERROR`.

Процедура `FCLOSE_ALL` закрывает все открытые файлы. Ее заголовок выглядит так:

```
PROCEDURE UTL_FILE.FCLOSE_ALL;
```

Эта процедура удобна тогда, когда программа открывает множество разных файлов, и вы хотите удостовериться в том, что ни один файл не останется открытым при ее завершении.

В программах, открывающих файлы, процедура `FCLOSE_ALL` может вызываться в обработчиках исключений; этот вызов гарантирует, что файлы будут закрыты и в случае аварийного завершения программы.

```
EXCEPTION
  WHEN OTHERS
  THEN
    UTL_FILE.FCLOSE_ALL;
    ... другие завершающие действия ...
END;
```

При закрытии файлов процедурой `FCLOSE_ALL` их файловые дескрипторы не помечаются как закрытые (иначе говоря, их поле `id` остается отличным от `NULL`). В результате любые вызовы `IS_OPEN` для таких дескрипторов будут возвращать `TRUE`, хотя вы и не сможете выполнять с этими файлами операции чтения/записи (без повторного открытия).

Чтение из файла

Процедура `UTL_FILE.GET_LINE` читает из открытого файла строку данных в заданный буфер. Заголовок процедуры выглядит так:

```
PROCEDURE UTL_FILE.GET_LINE
  (file IN UTL_FILE.FILE_TYPE,
   buffer OUT VARCHAR2);
```

Здесь *file* — дескриптор, полученный при вызове `FOPEN`, а *buffer* — буфер, в который читаются данные. Переменная, заданная для параметра *buffer*, должна быть достаточно большой для хранения всех данных вплоть до следующего символа новой строки или маркера конца файла. В противном случае PL/SQL инициирует исключение `VALUE_ERROR`. Символ-завершитель строки не включается в строку, записываемую в буфер.



Oracle предоставляет дополнительные версии `GET` для чтения данных `NVARCHAR2` (`GET_LINE_NCHAR`) и данных `RAW` (`GET_RAW`).

Пример использования `GET_LINE`:

```
DECLARE
  l_file UTL_FILE.FILE_TYPE;
  l_line VARCHAR2(32767);
BEGIN
  l_file := UTL_FILE.FOPEN ('TEMP_DIR', 'numlist.txt', 'R', max_linesize =>
32767);
  UTL_FILE.GET_LINE (l_file, l_line);
  DBMS_OUTPUT.PUT_LINE (l_line);
END;
```

`GET_LINE` читает данные только в строковую переменную, поэтому если в файле хранятся числа или даты, вам придется самостоятельно выполнить преобразование к локальной переменной соответствующего типа данных.

Исключения `GET_LINE`

Если `GET_LINE` пытается прочесть данные после конца файла, инициируется исключение `NO_DATA_FOUND`. Это же исключение инициируется в следующих ситуациях:

- при выполнении неявного (`SELECT INTO`) курсора, не возвращающего строк;
- при ссылке на неопределенную строку в коллекции PL/SQL;
- при чтении после конца двоичного файла (`BFILE`) с использованием `DBMS_LOB`.

Если вы выполняете несколько таких операций в одном блоке PL/SQL, вероятно, вам понадобится дополнительная логика для определения источника ошибки. Пример такого решения приведен в файле `who_did_that.sql` на сайте книги.

Удобная инкапсуляция для `GET_LINE`

Процедура `GET_LINE` проста и прямолинейна: она получает следующую строку из файла. Если файловый указатель уже находится на последней строке файла, `UTL_FILE.GET_LINE` не вернет никакие флаги, а инициирует исключение `NO_DATA_FOUND`. Это ухудшает структуру кода; возможно, вам стоит рассмотреть возможность инкапсуляции `GET_LINE` для решения этой проблемы, как объясняется в этом разделе.

Следующая программа читает каждую строку из файла и обрабатывает ее:

```
DECLARE
  l_file UTL_FILE.file_type;
  l_line VARCHAR2 (32767);
BEGIN
  l_file := UTL_FILE.FOPEN ('TEMP', 'names.txt', 'R');
```

```

LOOP
    UTL_FILE.get_line (l_file, l_line);
    process_line (l_line);
END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        UTL_FILE.fclose (l_file);
END;
```

Обратите внимание: этот простой цикл не содержит явных команд EXIT; он завершается неявно и с инициированием исключения сразу же после того, как UTL_FILE прочтает данные за концом файла. В подобных небольших блоках логика выглядит понятно, но представьте, что программа насчитывает сотни строк гораздо более сложного кода. Также допустим, что чтение содержимого файла — всего лишь один шаг в общем алгоритме. Если исключение завершает блок, остаток бизнес-логики необходимо разместить в разделе исключений (нежелательно) или же упаковать логику чтения файла в анонимном блоке BEGIN-END.

Меня такое решение не устраивает. Я не хочу программировать бесконечные циклы без команды EXIT; условие завершения не структурировано в самом цикле. Более того, ситуация достижения конца файла не является исключением; в конце концов, каждый файл когда-то да кончится. Почему я должен передавать управление в раздел исключений только из-за того, что я хочу полностью прочитать файл?

Полагаю, более удачный способ обработки конца файла заключается в построении программной «обертки» для GET_LINE, которая немедленно проверяет конец файла и возвращает логический признак (TRUE или FALSE). Этот подход продемонстрирован в следующей процедуре get_nextline:

```

/* Файл в Сети: getnext.sp */
PROCEDURE get_nextline (
    file_in IN UTL_FILE.FILE_TYPE
    , line_out OUT VARCHAR2
    , eof_out OUT BOOLEAN)
IS
BEGIN
    UTL_FILE.GET_LINE (file_in, line_out);
    eof_out := FALSE;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        line_out := NULL;
        eof_out := TRUE;
END;
```

Процедура get_nextline получает уже назначенный файловый дескриптор и возвращает два значения: строку текста (если она имеется) и логический признак (TRUE, если достигнут конец файла; FALSE в противном случае). Используя get_nextline, я могу прочитать содержимое файла в цикле с командой EXIT:

```

DECLARE
    l_file    UTL_FILE.file_type;
    l_line    VARCHAR2 (32767);
    l_eof     BOOLEAN;
BEGIN
    l_file := UTL_FILE.FOPEN ('TEMP', 'names.txt', 'R');

    LOOP
        get_nextline (l_file, l_line, l_eof);
```

```

        EXIT WHEN l_eof;
        process_line (l_line);
    END LOOP;
    UTL_FILE.fclose (l_file);
END;
```

С процедурой `get_nextline` достижение конца файла уже не рассматривается как исключительная ситуация. Я читаю строки из файла, пока он не закончится, после чего закрываю файл и выхожу из цикла. На мой взгляд, такая программа получается более понятной и прямолинейной.

Запись в файл

В отличие от относительно простой процедуры чтения, пакет `UTL_FILE` содержит несколько разных процедур, предназначенных для записи в файл:

- `UTL_FILE.PUT` — вывод данных в текущую строку открытого файла без присоединения завершителя. Либо используйте процедуру `NEW_LINE` для завершения текущей строки, либо выводите строку вместе с завершителем при помощи процедуры `PUT_LINE`.
- `UTL_FILE.NEW_LINE` — вставка одного (по умолчанию) или нескольких символов новой строки в текущей позиции.
- `UTL_FILE.PUT_LINE` — вывод в файл строки, за которой следует символ-завершитель, зависящий от платформы. Именно эта процедура чаще всего используется при работе с пакетом `UTL_FILE`.
- `UTL_FILE.PUTF` — вывод в файл до пяти строк в формате, определяемом форматной строкой (по аналогии с функцией `printf` языка C).
- `UTL_FILE.FFLUSH` — операции записи `UTL_FILE` обычно буферизуются; `FFLUSH` немедленно записывает содержимое буфера в файловую систему.

Все эти процедуры могут использоваться только в том случае, если файл был открыт в режиме `W` или `A`. Если файл был открыт только для чтения, исполняющее ядро инициирует исключение `UTL_FILE.INVALID_OPERATION`.



Oracle предоставляет дополнительные версии `PUT` для записи данных `NVARCHAR2` (`PUT_LINE_NCHAR`, `PUT_NCHAR`, `PUTF_NCHAR`) и данных `RAW` (`PUT_RAW`).

Рассмотрим пример использования `UTL_FILE.PUT_LINE`. Эта процедура записывает данные в файл, присоединяя за текстом символ новой строки. Заголовок `PUT_LINE` выглядит так:

```

PROCEDURE UTL_FILE.PUT_LINE (
    file IN UTL_FILE.FILE_TYPE
    ,buffer IN VARCHAR2
    ,autoflush IN BOOLEAN DEFAULT FALSE)
```

Здесь `file` — дескриптор, полученный при вызове `FOPEN`, а `buffer` — текст, записываемый в файл (длина не более 32 767); `autoflush` — `TRUE`, если вы хотите, чтобы эта строка немедленно была передана операционной системе.

Файл должен быть предварительно открыт перед вызовом `UTL_FILE.PUT_LINE`.

Пример использования `PUT_LINE` для вывода всех имен из таблицы в файл:

```

PROCEDURE names_to_file
IS
    fileid    UTL_FILE.file_type;
BEGIN
    fileid := UTL_FILE.FOPEN ('TEMP', 'names.dat', 'W');
```

```
FOR emprec IN (SELECT * FROM employee)
LOOP
    UTL_FILE.put_line (fileid, emprec.first_name || ' ' || emprec.last_name);
END LOOP;
```

```
    UTL_FILE.fclose (fileid);
END names_to_file;
```

Вызов PUT_LINE эквивалентен вызову PUT, за которым следует вызов NEW_LINE. Также он эквивалентен вызову PUTF с форматной строкой «%s\n» (см. описание PUTF в следующем разделе).

Запись отформатированного текста в файл

Как и PUT, процедура PUTF помещает данные в файл, но использует форматные данные (отсюда и суффикс F) для интерпретации разных элементов, помещаемых в файл. При вызове PUTF можно передавать от одного до пяти элементов данных. Заголовок процедуры выглядит так:

```
PROCEDURE UTL_FILE.putf
(
    file IN FILE_TYPE
  ,format IN VARCHAR2
  ,arg1 IN VARCHAR2 DEFAULT NULL
  ,arg2 IN VARCHAR2 DEFAULT NULL
  ,arg3 IN VARCHAR2 DEFAULT NULL
  ,arg4 IN VARCHAR2 DEFAULT NULL
  ,arg5 IN VARCHAR2 DEFAULT NULL);
```

Здесь *file* — дескриптор, полученный при вызове FOPEN, а *format* — строка, определяющая формат элементов в файле (см. ниже); параметры *argN* определяют необязательные строковые аргументы (от одного до пяти).

Форматная строка позволяет подставить значения *argN* прямо в текст, записываемый в файл. Кроме «шаблонного» (то есть литерального) текста форматная строка может содержать следующие служебные комбинации:

- %s — приказывает PUTF поместить соответствующий элемент в файл. Форматная строка может содержать до пяти комбинаций %s, потому что PUTF получает до пяти элементов.
- \n — приказывает PUTF поместить в файл символ новой строки. Количество вхождений \n в форматную строку не ограничено.

Комбинации %s заменяются строками аргументов в указанном порядке. Если при вызове передано недостаточно значений для подстановки на место всех форматных комбинаций, %s просто удаляется из строки перед записью в файл.

Следующий пример демонстрирует использование форматных строк. Допустим, содержимое файла должно выглядеть так:

```
Employee: Steven Feuerstein
Soc Sec #: 123-45-5678
Salary: $1000
```

Задача решается одним вызовом PUTF:

```
UTL_FILE.PUTF
(
    file_handle, 'Employee: %s\nSoc Sec #: %s\nSalary: %s\n',
    'Steven Feuerstein',
    '123-45-5678',
    TO_CHAR (:employee.salary, '$9999'));
```

Если вам потребуется вывести более пяти элементов данных, просто вызовите PUTF дважды для завершения работы.

Копирование файлов

Процедура `UTL_FILE.FCOPY` позволяет легко скопировать содержимое одного файла в другой. Например, в следующем фрагменте в архивном каталоге создается резервная копия файла из каталога разработки:

```
DECLARE
    file_suffix    VARCHAR2 (100)
                  := TO_CHAR (SYSDATE, 'YYYYMMDDHH24MISS');
BEGIN
    -- Копирование всего файла ...
    UTL_FILE.FCOPY (
        src_location      => 'DEVELOPMENT_DIR',
        src_filename      => 'archive.zip',
        dest_location     => 'ARCHIVE_DIR',
        dest_filename     => 'archive'
                        || file_suffix
                        || '.zip'
    );
END;
```

Процедура `FCOPY` также может использоваться для копирования *части* файла. Два дополнительных параметра задают номера начальной и конечной строки блока, копируемого из файла. Допустим, в текстовом файле хранятся имена победителей ежемесячного конкурса знатоков PL/SQL, начавшегося в январе 2008 года. Мы хотим скопировать все имена, относящиеся к 2009 году, в другой файл. В этом нам помогут пятый и шестой аргументы `FCOPY`:

```
DECLARE
    c_start_year CONSTANT PLS_INTEGER := 2008;
    c_year_of_interest CONSTANT PLS_INTEGER := 2009;
    l_start PLS_INTEGER;
    l_end PLS_INTEGER;
BEGIN
    l_start := (c_year_of_interest - c_start_year)*12 + 1;
    l_end := l_start + 11;

    UTL_FILE.FCOPY (
        src_location      => 'WINNERS_DIR',
        src_filename      => 'names.txt',
        dest_location     => 'WINNERS_DIR',
        dest_filename     => 'names2008.txt',
        start_line        => l_start,
        end_line          => l_end
    );
END;
```

Удобная инкапсуляция `UTL_FILE.FCOPY` позволит задавать начальную и конечную строки вместо номеров строк. При желании вы можете реализовать такую программу самостоятельно (в файле `infile.sf` на сайте книги приведена реализация «INSTR для файлов», которая может стать хорошей отправной точкой).

Удаление файлов

В Oracle9i Release 2 и последующих версиях для удаления файлов используется процедура `UTL_FILE.REMOVE`. Заголовок процедуры выглядит так:

```
PROCEDURE UTL_FILE.REMOVE (
    location IN VARCHAR2,
    filename IN VARCHAR2);
```

Например, в следующем фрагменте `UTL_FILE.REMOVE` удаляет архивный файл из предыдущего примера:

```
BEGIN
  UTL_FILE.FREMOVE ('DEVELOPMENT_DIR', 'archive.zip');
END;
```

Вы задаете путь и имя файла, а UTL_FILE *пытается* его удалить. А если при удалении возникнут проблемы? Будет инициировано одно из следующих исключений.

| Исключение | Описание |
|---------------------------|--|
| UTL_FILE.invalid_path | Недействительный дескриптор |
| UTL_FILE.invalid_filename | Файл не найден или имя файла равно NULL |
| UTL_FILE.file_open | Файл уже открыт для записи/присоединения |
| UTL_FILE.access_denied | Отказано в доступе к объекту каталога |
| UTL_FILE.remove_failed | Сбой при удалении файла |

Иначе говоря, UTL_FILE инициирует исключение при попытке удаления несуществующего файла или при отсутствии привилегий, необходимых для удаления файла. Многие программы удаления файлов в других языках (например, `File.delete` в Java) возвращают код состояния, который информирует вас о результате попытки удаления. Если вы предпочитаете этот способ, используйте (или скопируйте) программу `fileIO.FREMOVE` из файла `fileIO.pkg` на сайте книги.

Переименование и перемещение файлов

Операции копирования и удаления объединяются в вызове процедуры `UTL_FILE.RENAME`. Эта удобная утилита позволяет либо переименовать файл в том же каталоге, либо изменить как имя, так и местонахождение (то есть фактически переместить файл).

Заголовок `FRENAME` выглядит так:

```
PROCEDURE UTL_FILE.frename (
  src_location   IN VARCHAR2,
  src_filename   IN VARCHAR2,
  dest_location  IN VARCHAR2,
  dest_filename  IN VARCHAR2,
  overwrite      IN BOOLEAN DEFAULT FALSE);
```

Программа может инициировать одно из следующих исключений.

| Исключение | Описание |
|---------------------------|---|
| UTL_FILE.invalid_path | Недействительный дескриптор |
| UTL_FILE.invalid_filename | Файл не найден или имя файла равно NULL |
| UTL_FILE.access_denied | Отказано в доступе к объекту каталога |
| UTL_FILE.remove_failed | Сбой при удалении файла |

Интересный пример применения `FRENAME` встречается в пакете `fileIO.pkg`. Процедура `chgext` изменяет расширение заданного файла.

Получение атрибутов файла

Иногда требуется получить информацию о конкретном файле. Сколько места занимает файл? Существует ли он? Какой размер блока используется для хранения файла? Ответы уже не являются секретами, которые могут быть разрешены только с помощью команды операционной системы (или в случае длины файла — пакета `DBMS_LOB`), как в ранних версиях Oracle. Всю эту информацию можно получить за один вызов процедуры `UTL_FILE.FGETATTR`.

Заголовок `FGETATTR` выглядит так:

```

PROCEDURE UTL_FILE.FGETATTR (
  location   IN VARCHAR2,
  filename   IN VARCHAR2,
  fexists    OUT BOOLEAN,
  file_length OUT NUMBER,
  block_size OUT BINARY_INTEGER);

```

Таким образом, чтобы использовать эту программу, вы должны объявить три переменные для хранения логического признака (существует ли файл?), длины файла и размера блока. Пример использования:

```

DECLARE
  l_fexists    BOOLEAN;
  l_file_length PLS_INTEGER;
  l_block_size  PLS_INTEGER;
BEGIN
  UTL_FILE.FGETATTR (
    location   => 'DEVELOPMENT_DIR',
    filename   => 'bigpkg.pkg',
    fexists    => l_fexists,
    file_length => l_file_length,
    block_size => l_block_size
  );
  ...
END;

```

Такой интерфейс весьма неудобен. Даже если вас интересует только длина файла, все равно придется объявлять все переменные, получать длину, а затем работать с полученным значением. Возможно, для работы с FGETATTR стоит создать собственные функции-«обертки», которые позволяют получить отдельное значение:

```

FUNCTION fileIO.length (
  location_in IN VARCHAR2,
  file_in     IN VARCHAR2
)
RETURN PLS_INTEGER;

```

или:

```

FUNCTION fileIO.fexists (
  location_in IN VARCHAR2,
  file_in     IN VARCHAR2
)
RETURN BOOLEAN;

```

В этом случае вам не приходится объявлять ненужные переменные, а программный код становится проще и лаконичнее.

Отправка электронной почты

За прошедшие годы отправка электронной почты из хранимых процедур в Oracle постепенно упрощалась. Короткий пример:

```

/* Требуется Oracle10g и выше */
BEGIN
  UTL_MAIL.send(
    sender      => 'me@mydomain.com'
  ,recipients  => 'you@yourdomain.com'
  ,subject     => 'API for sending email'
  ,message     =>
'Dear Friend:
This is not spam. It is a mail test.
Mailfully Yours,
Bill'
  );
END;

```


При выполнении этого блока Oracle пытается отправить сообщение с использованием хоста SMTP (Simple Mail Transfer Protocol), настроенного в файле инициализации (см. следующий раздел). Заголовок `UTL_MAIL.SEND` выглядит так:

```
PROCEDURE send(sender      IN VARCHAR2,
               recipients  IN VARCHAR2,
               cc          IN VARCHAR2 DEFAULT NULL,
               bcc         IN VARCHAR2 DEFAULT NULL,
               subject     IN VARCHAR2 DEFAULT NULL,
               message     IN VARCHAR2 DEFAULT NULL,
               mime_type   IN VARCHAR2
                  DEFAULT 'text/plain; charset=us-ascii',
               priority    IN PLS_INTEGER DEFAULT 3);
```

Большинство параметров понятно без пояснений. Один нетривиальный совет по использованию: чтобы отправить сообщение нескольким получателям (в том числе и в полях `cc` и `bcc`), разделите адреса запятыми:

```
recipients => 'you@yourdomain.com, him@hisdomain.com'
```

Если вы работаете в более ранней версии Oracle или хотите в большей степени контролировать процесс отправки, вы, как и прежде, можете использовать пакет `UTL_SMTP` — процедура получается чуть более сложной, но тем не менее работоспособной. Если отправка почты должна программироваться на еще более низком уровне, используйте `UTL_TCP`, внешнюю процедуру или хранимую процедуру Java.

Предварительная настройка

К сожалению, не все версии Oracle предоставляют готовый сервис отправки электронной почты. Встроенный пакет `UTL_SMTP` включается в установку по умолчанию, поэтому он обычно работает сразу. Но если вы работаете в Oracle11g Release 2, вам придется еще немного повозиться с настройкой безопасности.

Начиная с Oracle10g пакет `UTL_MAIL` не включается в стандартную установку Oracle. Чтобы подготовить `UTL_MAIL` к использованию, ваш администратор должен:

1. Присвоить значение параметру инициализации `SMTP_OUT_SERVER`. В Oracle10g Release 2 и выше это делается примерно так:

```
ALTER SYSTEM SET SMTP_OUT_SERVER = 'mailhost';
```

В Oracle10g Release 1 для настройки параметра приходилось вручную редактировать `pfile`. Строка содержит имена одного или нескольких хостов (разделенных запятыми); `UTL_MAIL` последовательно перебирает их, пока не найдет подходящий.

2. После настройки параметра *перезагрузить сервер базы данных*, чтобы изменения вступили в силу. Удивительно, но факт.
3. Выполнить следующие сценарии с правами SYS:

```
@$ORACLE_HOME/rdbms/admin/utlmail.sql
@$ORACLE_HOME/rdbms/admin/prvtmail.plb
```

4. Предоставить привилегии выполнения `UTL_MAIL` тем, кто будет пользоваться этим пакетом:

```
GRANT EXECUTE ON UTL_MAIL TO SCOTT;
```

Настройка сетевой безопасности

В Oracle11g Release 2 администратору базы данных придется проделать еще одно действие для любого пакета, выполняющего внешние сетевые вызовы, — к их числу относятся и `UTL_SMTP` с `UTL_MAIL`. Администратор должен создать *спусок ACL* (Access Control List),

включить в него имя пользователя или роль и предоставить списку привилегию сетевого уровня. Простая заготовка ACL для этой цели может выглядеть так:

```
BEGIN
  DBMS_NETWORK_ACL_ADMIN.CREATE_ACL (
    acl          => 'mail-server.xml'
    ,description => 'Permission to make network connections to mail server'
    ,principal   => 'SCOTT' /* Имя пользователя или роль */
    ,is_grant    => TRUE
    ,privilege    => 'connect'
  );

  DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL (
    acl          => 'mail-server.xml'
    ,host        => 'my-STMP-servername'
    ,lower_port  => 25 /* Сетевой порт SMTP по умолчанию */
    ,upper_port  => NULL /* Открывается только порт 25 */
  );
END;
```

В наши дни сетевому администратору также иногда приходится вносить изменения в настройку брандмауэра, чтобы разрешить исходящие подключения через порт 25 с сервера базы данных. Возможно, администратору электронной почты тоже придется задать дополнительные разрешения!

Отправка короткого текстового сообщения

В предыдущем разделе было показано, как отправить текстовое сообщение средствами UTL_MAIL. Если вы используете UTL_SMTP, вашей программе придется взаимодействовать с почтовым сервером на более низком уровне: она должна открыть подключение, сформировать заголовки, отправить тело сообщения и (в идеале) проанализировать возвращаемые значения. На рис. 22.1 изображена схема взаимодействия между почтовым сервером и почтовым клиентом PL/SQL send_mail_via_util_smtp.

Код этой простой хранимой процедуры:

```
/* Файл в Сети: send_mail_via_util_smtp.sp */
1  PROCEDURE send_mail_via_util_smtp
2  ( sender IN VARCHAR2
3  , recipient IN VARCHAR2
4  , subject IN VARCHAR2 DEFAULT NULL
5  , message IN VARCHAR2
6  , mailhost IN VARCHAR2 DEFAULT 'mailhost'
7  )
8  IS
9  mail_conn UTL_SMTP.connection;
10  crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
11  smtp_tcpip_port CONSTANT PLS_INTEGER := 25;
12  BEGIN
13  mail_conn := UTL_SMTP.OPEN_CONNECTION(mailhost, smtp_tcpip_port);
14  UTL_SMTP.HELO(mail_conn, mailhost);
15  UTL_SMTP.MAIL(mail_conn, sender);
16  UTL_SMTP.RCPT(mail_conn, recipient);
17  UTL_SMTP.DATA(mail_conn, SUBSTR(
18  'Date: ' || TO_CHAR(SYSTIMESTAMP, 'Dy, dd Mon YYYY HH24:MI:SS TZHTZM')
19  || crlf || 'From: ' || sender || crlf
20  || 'Subject: ' || subject || crlf
21  || 'To: ' || recipient || crlf
22  || message
23  , 1, 32767));
24
25  UTL_SMTP.QUIT(mail_conn);
26  END;
```

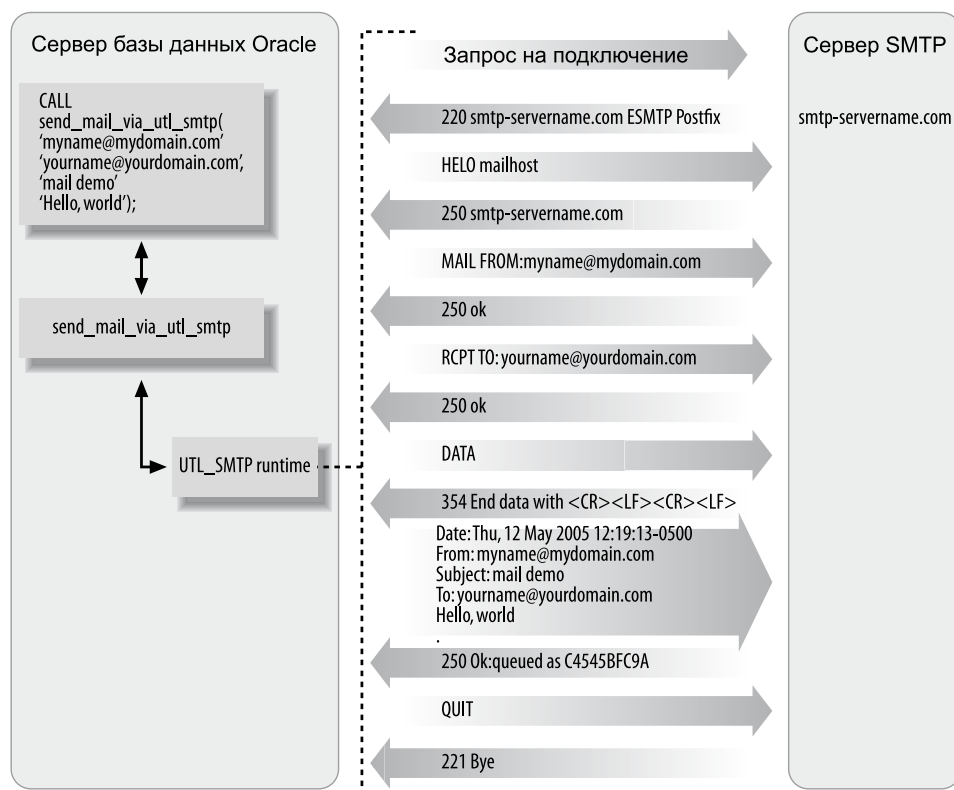


Рис. 22.1. Схема взаимодействия между почтовым клиентом PL/SQL и сервером SMTP

Ключевые аспекты этого кода перечислены в следующей таблице.

| Строки | Описание |
|--------|---|
| 9 | В программе определяется переменная, представляющая подключение — запись типа <code>UTL_SMTP.connection</code> |
| 10 | Согласно почтовым стандартам Интернета, все заголовки строк должны заканчиваться комбинацией символов возврата курсора и перевода строки (см. строки 19–21) |
| 14–25 | Эти строки передают инструкции серверу SMTP и формируют почту в стандартном формате, которую ожидает получить сервер |
| 18 | Данные типа <code>SYSTIMESTAMP</code> (введенного в Oracle9i) используются для получения доступа к данным часового пояса |

Из строк 17–23 видно, что эта процедура не может отправить сообщение, у которого длина части `DATA` превышает 32 767 байт (максимальная длина переменных PL/SQL). Пакет `UTL_SMTP` позволяет отправлять и более длинные сообщения, но вам придется организовать потоковую запись данных с использованием многократных вызовов `UTL_SMTP.WRITE_DATA`.



Большинство почтовых программ ограничивает каждую строку текста 78 символами и двумя символами-завершителями. В общем случае рекомендуется не включать в каждую строку текста более 998 символов помимо символов возврата курсора/перевода строки (1000 байт, если считать завершители CR/LF). Не превышайте предел в 1000 байт, если вы не уверены в том, что ваш сервер реализует расширения SMTP «Service Extension».

Включение «удобных» имен в адреса электронной почты

Если вызвать приведенную выше процедуру следующим образом:

```
BEGIN
  send_mail_via_utl_smtp('myname@mydomain.com',
    'yourname@yourdomain.com', 'mail demo', NULL);
END;
```

«видимые» заголовки сообщения, генерируемые строками 17–21, будут выглядеть примерно так:

```
Date: Wed, 23 Mar 2005 17:14:30 -0600
From: myname@mydomain.com
Subject: mail demo
To: yourname@yourdomain.com
```

Люди (и многие программы для борьбы со спамом) предпочитают видеть в заголовках реальные имена в форме:

```
Date: Wed, 23 Mar 2005 17:14:30 -0600
From: Bob Swordfish <myname@mydomain.com>
Subject: mail demo
To: "Scott Tiger, Esq." <yourname@yourdomain.com>
```

Конечно, такое изменение можно внести несколькими способами; пожалуй, самое элегантное решение основано на разборе параметров отправителя и получателя. Именно такое решение Oracle использует в UTL_MAIL. Итак, например, UTL_MAIL.SEND можно вызывать с адресами следующего вида:

```
["удобочитаемое имя"] <адрес_электронной_почты>
```

как в следующем примере:

```
BEGIN
  UTL_MAIL.send('Bob Swordfish <myname@mydomain.com>',
    '"Scott Tiger, Esq." <yourname@yourdomain.com>',
    subject=>'mail demo');
END;
```

Но следует учитывать, что пакет Oracle также добавляет информацию о наборе символов, поэтому приведенный код генерирует заголовок следующего вида:

```
Date: Sat, 24 Jan 2005 17:47:00 -0600 (CST)
From: Bob Swordfish <me@mydomain.com>
To: Scott Tiger, Esq. <you@yourdomain.com>
Subject: =?WINDOWS-1252?Q?mail=20demo?=
```

И хотя для пользователей, привыкших к ASCII, такая запись может показаться странной, в области интернет-стандартов она вполне допустима; разумный почтовый клиент все равно должен интерпретировать информацию кодировки (вместо того, чтобы просто выводить ее). Одной из очевидных модификаций процедуры `send_mail_via_utl_smtp` может стать добавление параметров для удобочитаемых имен (или преобразование существующих параметров в запись).

Отправка текстового сообщения произвольной длины

Пакет UTL_MAIL удобен, но при отправке текстового сообщения, длина которого превышает 32 767 байт, он вам не поможет. Чтобы обойти это ограничение, можно изменить процедуру `send_mail_via_utl_smtp` так, чтобы параметр `message` имел тип данных CLOB. Также придется внести ряд других изменений:

```
/* Файл в Сети: send_clob.sp */
PROCEDURE send_clob_thru_email (
  sender    IN VARCHAR2
```

```

, recipient IN VARCHAR2
, subject   IN VARCHAR2 DEFAULT NULL
, MESSAGE   IN CLOB
, mailhost  IN VARCHAR2 DEFAULT 'mailhost'
)
IS
mail_conn      UTL_SMTP.connection;
crlf           CONSTANT VARCHAR2 (2) := CHR (13) || CHR (10);
smtp_tcpip_port CONSTANT PLS_INTEGER := 25;
pos           PLS_INTEGER := 1;
bytes_o_data   CONSTANT PLS_INTEGER := 32767;
offset         PLS_INTEGER := bytes_o_data;
msg_length     CONSTANT PLS_INTEGER := DBMS_LOB.getlength (MESSAGE);
BEGIN
mail_conn := UTL_SMTP.open_connection (mailhost, smtp_tcpip_port);
UTL_SMTP.helo (mail_conn, mailhost);
UTL_SMTP.mail (mail_conn, sender);
UTL_SMTP.rcpt (mail_conn, recipient);

UTL_SMTP.open_data (mail_conn);
UTL_SMTP.write_data (
    mail_conn
    , 'Date: '
    || TO_CHAR (SYSTIMESTAMP, 'Dy, dd Mon YYYY HH24:MI:SS TZHTZM')
    || crlf
    , 'From: '
    || sender
    || crlf
    , 'Subject: '
    || subject
    || crlf
    , 'To: '
    || recipient
    || crlf
);
WHILE pos < msg_length
LOOP
    UTL_SMTP.write_data (mail_conn, DBMS_LOB.SUBSTR (MESSAGE, offset, pos));
    pos := pos + offset;
    offset := LEAST (bytes_o_data, msg_length - offset);
END LOOP;
UTL_SMTP.close_data (mail_conn);

UTL_SMTP.quit (mail_conn);
END send_clob_thru_email;

```

Вызовы `open_data`, `write_data` и `close_data` позволяют передать произвольное количество байтов почтовому серверу (до максимального размера почтового сообщения, установленного сервером). Учтите, что в этом коде делается одно серьезное допущение: предполагается, что данные CLOB были разбиты на строки правильной длины.

Теперь давайте посмотрим, как присоединить файл к сообщению.

Отправка сообщения с коротким вложением

Исходный стандарт электронной почты требовал, чтобы все сообщения состояли только из 7-разрядных ASCII-символов¹. Но как известно, в сообщениях могут присутствовать вложения, которые обычно хранятся в двоичном, а не в текстовом формате. Как передать

¹ Современные почтовые программы обычно поддерживают пересылку 8-разрядных символов с использованием расширения SMTP 8BITMIME. Чтобы проверить ее поддержку, воспользуйтесь директивой SMTP EHLO.

двоичный файл в сообщении ASCII? Обычно для пересылки вложений используются почтовые расширения MIME¹ (Multipurpose Internet Mail Extensions) в сочетании со схемой преобразования двоичных данных в ASCII base64. Следующий пример показывает, как происходит пересылка небольшого двоичного файла:

```
Date: Wed, 01 Apr 2009 10:16:51 -0600
From: Bob Swordfish <my@myname.com>
MIME-Version: 1.0
To: Scott Tiger <you@yourname.com>
Subject: Attachment demo
Content-Type: multipart/mixed;
    boundary="-----060903040208010603090401"

This is a multi-part message in MIME format.
-----060903040208010603090401
Content-Type: text/plain; charset=us-ascii; format=fixed
Content-Transfer-Encoding: 7bit
```

Dear Scott:

I'm sending a gzipped file containing the text of the first paragraph. Hope you like it.

```
Bob
-----060903040208010603090401
Content-Type: application/x-gzip; name="hugo.txt.gz"
```

```
Content-Transfer-Encoding: base64
Content-Disposition: inline; filename="hugo.txt.gz"
H4sICDh/TUICA2xlc21pcy50eHQAPY5BDoJAEATvvqI/AJGDxjMaowcesbKN0wmZITsshf7
DdGD105Vpe+K5tQc0Jm6sGScU8gjvbrmoG8Tr1qhLtSCbs3CEa/gaMwTTbABF3kqa9z42+dE
RXhYmeHcpHmtBlmIoBEpREyZlpERTjB/aUSxns5/Ci7ac/u0P9a7Dw4FECSdAAAA
-----060903040208010603090401--
```

Хотя большая часть кода может быть стандартной, однако при генерировании таких сообщений необходимо следить за множеством технических деталей. К счастью, при отправке «небольших» вложений (с длиной менее 32 767 байт) из Oracle10g и выше на помощь приходит пакет UTL_MAIL. В следующем примере используется процедура UTL_MAIL.SEND_ATTACH_VARCHAR2, которая отправляет вложения в текстовом виде.

Отправка приведенного выше сообщения выполняется следующим образом:

```
DECLARE
    b64 VARCHAR2(512) := 'H4sICDh/TUICA2xlc21...'; -- См. выше
    txt VARCHAR2(512) := 'Dear Scott: ...'; -- См. выше
BEGIN
    UTL_MAIL.send_attach_varchar2(
        sender => 'my@myname.com'
        ,recipients => 'you@yourname.com'
        ,message => txt
        ,subject => 'Attachment demo'
        ,att_mime_type => 'application/x-gzip'
        ,attachment => b64
        ,att_inline => TRUE
        ,att_filename => 'hugo.txt.gz'
    );
END;
```

¹ См. RFC 2045, 2046, 2047, 2048 и 2049, а также обновления 2184, 2231, 2646 и 3023.

Новые параметры описаны в следующей таблице.

| Параметры | Описание |
|---------------|---|
| att_mime_type | Обозначение типа |
| att_inline | Флаг, указывающий почтовому клиенту, как должно отображаться вложение: в теле сообщения (TRUE) или отдельно (FALSE) |
| att_filename | Имя вложенного файла, заданное отправителем |

Типы MIME не могут выбираться произвольно; они, как и многие другие технические аспекты Интернета, определяются комитетом IANA (Internet Assigned Numbers Authority).

Среди часто используемых типов содержимого MIME можно выделить `text/plain`, `multipart/mixed`, `text/html`, `application/pdf` и `application/msword`. Полный список приведен на странице IANA по адресу <http://www.iana.org/assignments/media-types/>.

Вероятно, вы заметили, что для присоединения к сообщению файла в кодировке base64 пришлось основательно потрудиться. Давайте подробнее рассмотрим действия, необходимые для преобразования двоичного файла в объект, пригодный для пересылки.

Отправка небольшого файла во вложении

Чтобы преобразовать небольшой двоичный файл в объект, который можно отправить в сообщении электронной почты, можно прочитать содержимое файла в переменную RAW и воспользоваться процедурой `UTL_MAIL.SEND_ATTACH_RAW`. Oracle преобразует двоичные данные в кодировку base64 и формирует необходимые директивы MIME. Скажем, пересылка файла `/tmp/hugo.txt.gz` (размером менее 32 767 байт) может быть выполнена следующим образом:

```
/* Файл в Сети: send_small_file.sql */
CREATE OR REPLACE DIRECTORY tmpdir AS '/tmp'
/
DECLARE
    the_file BFILE := BFILENAME('TMPDIR', 'hugo.txt.gz');
    rawbuf RAW(32767);
    amt PLS_INTEGER := 32767;
    offset PLS_INTEGER := 1;
BEGIN
    DBMS_LOB.fileopen(the_file, DBMS_LOB.file_readonly);
    DBMS_LOB.read(the_file, amt, offset, rawbuf);
    UTL_MAIL.send_attach_raw
(
    sender => 'my@myname.com'
  ,recipients => 'you@yourname.com'
  ,subject => 'Attachment demo'
  ,message => 'Dear Scott...'
  ,att_mime_type => 'application/x-gzip'
  ,attachment => rawbuf
  ,att_inline => TRUE
  ,att_filename => 'hugo.txt.gz'
);

    DBMS_LOB.close(the_file);
END;
```

Если пакет `UTL_MAIL` недоступен, используйте инструкции, приведенные в следующем разделе.

Вложение файла произвольного размера

Для отправки вложения большого размера можно воспользоваться традиционным пакетом UTL_SMTP; вложение преобразуется в кодировку base64 средствами встроенного пакета Oracle UTL_ENCODE. Следующая процедура отправляет BFILE с коротким текстовым сообщением:

```

/* Файл в Сети: send_bfile.sp */
1  PROCEDURE send_bfile
2    ( sender IN VARCHAR2
3      ,recipient IN VARCHAR2
4      ,subject IN VARCHAR2 DEFAULT NULL
5      ,message IN VARCHAR2 DEFAULT NULL
6      ,att_bfile IN OUT BFILE
7      ,att_mime_type IN VARCHAR2
8      ,mailhost IN VARCHAR2 DEFAULT 'mailhost'
9    )
10  IS
11    crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
12    smtp_tcpip_port CONSTANT PLS_INTEGER := 25;
13    bytes_per_read CONSTANT PLS_INTEGER := 23829;
14    boundary CONSTANT VARCHAR2(78) := '-----5e9i1BxFQrgl9c0gs90-----';
15    encapsulation_boundary CONSTANT VARCHAR2(78) := '--' || boundary;
16    final_boundary CONSTANT VARCHAR2(78) := '--' || boundary || '--';
17
18    mail_conn UTL_SMTP.connection;
19    pos PLS_INTEGER := 1;
20    file_length PLS_INTEGER;
21
22    diralias VARCHAR2(30);
23    bfile_filename VARCHAR2(512);
24    lines_in_bigbuf PLS_INTEGER := 0;
25
26    PROCEDURE writedata (str IN VARCHAR2, crlfs IN PLS_INTEGER DEFAULT 1)
27    IS
28      BEGIN
29        UTL_SMTP.write_data(mail_conn, str || RPAD(crlf, 2 * crlfs, crlf));
30      END;
31
32  BEGIN
33    DBMS_LOB.fileopen(att_bfile, DBMS_LOB.LOB_READONLY);
34    file_length := DBMS_LOB.getlength(att_bfile);
35
36    mail_conn := UTL_SMTP.open_connection(mailhost, smtp_tcpip_port);
37    UTL_SMTP.helo(mail_conn, mailhost);
38    UTL_SMTP.mail(mail_conn, sender);
39    UTL_SMTP.rcpt(mail_conn, recipient);
40
41    UTL_SMTP.open_data(mail_conn);
42    writedata('Date: ' || TO_CHAR(SYSTIMESTAMP,
43      'Dy, dd Mon YYYY HH24:MI:SS TZHTZM') || crlf
44      || 'MIME-Version: 1.0' || crlf
45      || 'From: ' || sender || crlf
46      || 'Subject: ' || subject || crlf
47      || 'To: ' || recipient || crlf
48      || 'Content-Type: multipart/mixed; boundary="' || boundary || '"', 2);
49
50    writedata(encapsulation_boundary);
51    writedata('Content-Type: text/plain; charset=ISO-8859-1; format=flowed');
52    writedata('Content-Transfer-Encoding: 7bit', 2);
53    writedata(message, 2);
54
55    DBMS_LOB.filegetname(att_bfile, diralias, bfile_filename);
56    writedata(encapsulation_boundary);

```



```

57 writedata('Content-Type: '
58 || att_mime_type || '; name="' || bfile_filename || '');
59 writedata('Content-Transfer-Encoding: base64');
60 writedata('Content-Disposition: attachment; filename="'
61 || bfile_filename || '", 2);
62
63 WHILE pos < file_length
64 LOOP
65     writedata(UTL_RAW.cast_to_varchar2(
66         UTL_ENCODE.base64_encode
67         DBMS_LOB.substr(att_bfile, bytes_per_read, pos))), 0);
68     pos := pos + bytes_per_read;
69 END LOOP;
70
71 writedata(crlf || crlf || final_boundary);
72
73 UTL_SMTP.close_data(mail_conn);
74 UTL_SMTP.QUIT(mail_conn);
75 DBMS_LOB.CLOSE(att_bfile);
76 END;

```

Ключевые моменты этого кода перечислены в следующей таблице.

| Строки | Описание |
|--------|--|
| 13 | Константа определяет, сколько байтов файла будет читаться одной операцией чтения (см. строку 67). По соображениям производительности значение должно быть как можно большим. Известно, что UTL_ENCODE.BASE64_ENCODE генерирует строки длиной 64 символа, а алгоритм base64 преобразует каждые 3 байта двоичных данных в 4 байта символьных данных. Прибавьте 2 байта CRLF на строку текста base64, и вы получите наибольшую возможную длину читаемого блока в 23 829 байт ($\text{TRUNC}((0.75*64)*(32767/(64+2))-1)$) |
| 14–16 | Граничная строка может использоваться повторно. Но если вы захотите создать сообщение с вложенными объектами MIME, на разных уровнях вложенности должны использоваться разные граничные строки |
| 26–30 | Вспомогательная процедура, которая немного упрощает исполняемый раздел. Параметр crlfs определяет количество завершителей CRLF, присоединяемых к файлу (обычно 0, 1 или 2) |
| 55 | Вместо того чтобы передавать дополнительный аргумент с именем файла, его можно извлечь непосредственно из BFILE |
| 63–69 | Основной код программы читает часть файла, преобразует ее в кодировку base64 и отправляет данные по почтовому подключению до достижения предела в 32 767 байт |

Да, нам когда-то тоже казалось, что отправка электронной почты — простое дело. К тому же эта процедура не обладает особой гибкостью: она позволяет отправить одну текстовую часть с вложением одного файла. Но по крайней мере вы можете взять ее за основу и доработать с учетом потребностей вашего собственного приложения.

И еще одно замечание по поводу построения правильно структурированных сообщений электронной почты: вместо того, чтобы зарываться в документацию RFC, попробуйте запустить почтового клиента, которого вы используете в повседневной работе, отправьте себе сообщение в форме, которую вы пытаетесь сгенерировать, а затем просмотрите исходный текст сообщения. Я столько раз проделывал это во время работы над этим разделом книги! Учтите, что некоторые почтовые клиенты (прежде всего Microsoft Outlook) не предоставляют средств для полного просмотра низкоуровневого текста.

Работа с данными в Интернете (HTTP)

Допустим, вы хотите загрузить данные с сайта своего бизнес-партнера. Существует много способов выборки веб-страниц.

- «Ручная выборка», то есть ввод нужного адреса в браузере.
- Использование сценарного языка — например, Perl. Кстати говоря, этот язык содержит много всевозможных вспомогательных средств, упрощающих интерпретацию загруженных данных.
- Использование утилиты командной строки — например, GNU *wget*.
- Использование встроенного пакета Oracle UTL_HTTP.

Так как книга посвящена PL/SQL, мы будем рассматривать последний метод.

В Oracle11g Release 2 и выше вам для этого придется создать сетевой список ACL для настройки исходящих подключений к нужным удаленным хостам (см. предыдущий раздел).

Начнем с относительно простого способа программирования выборки данных с веб-страницы — «нарезки» веб-страницы на фрагменты, сохраняемые в элементах массива. Этот способ использовался в Oracle до поддержки CLOB.

Фрагментная загрузка страницы

Одна из первых процедур пакета UTL_HTTP загружала веб-страницу в последовательные элементы ассоциативного массива:

```
DECLARE
    page_pieces UTL_HTTP.html_pieces; -- array of VARCHAR2(2000)
BEGIN
    page_pieces := UTL_HTTP.request_pieces(url => 'http://www.oreilly.com/');
END;
```

Работать с данными в таком формате было неудобно, потому что границы 2000-байтовых фрагментов совершенно не совпадали с текстом страницы. Таким образом, если алгоритм разбора данных базировался на построчной обработке, строки приходилось читать из массива и собирать заново. Кроме того, по данным Oracle, не все (не завершающие) фрагменты дополняются до 200 байт; алгоритм Oracle не использует границы строк как точки разбиения; и максимальное количество фрагментов равно 32 767.

Впрочем, даже если этот алгоритм удовлетворял потребностям программиста, с некоторыми сайтами приведенный код не работал. Например, некоторые сайты отказывались предоставлять данные такому сценарию, потому что используемый по умолчанию Oracle заголовок HTTP был неизвестен веб-серверу. В частности, заголовок «User-Agent» содержит текстовую строку с информацией о браузере, используемом (или эмулируемом) клиентом, а многие сайты предоставляют контент, ориентированный на конкретные браузеры. По умолчанию программы Oracle не передавали заголовок «User-Agent», который мог выглядеть примерно так:

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)

Дополнительная передача этого заголовка увеличивает сложность кода, потому что программисту приходилось работать на более низком уровне: он должен был инициировать «запрос», передать заголовок, получить «ответ» и загружать страницу в цикл:

```
DECLARE
    req UTL_HTTP.req;      -- "Объект запроса" (на самом деле запись PL/SQL)
    resp UTL_HTTP.resp;    -- "Объект ответа" (тоже запись PL/SQL)
    buf VARCHAR2(32767);   --- буфер для хранения данных страницы
BEGIN
    req := UTL_HTTP.begin_request('http://www.oreilly.com/',
        http_version => UTL_HTTP.http_version_1_1);
    UTL_HTTP.set_header(req, 'User-Agent'
        , 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)');
    resp := UTL_HTTP.get_response(req);
```

```

BEGIN
  LOOP
    UTL_HTTP.read_text(resp, buf);
    -- Обработка содержимого buf (например, сохранение в массиве)
  END LOOP;
EXCEPTION
  WHEN UTL_HTTP.end_of_body
  THEN
    NULL;
END;
UTL_HTTP.end_response(resp);
END;

```

Центральное место в этом коде занимает следующая встроенная процедура:

```

PROCEDURE UTL_HTTP.read_text(
  r IN OUT NOCOPY UTL_HTTP.resp,
  data OUT NOCOPY VARCHAR2 CHARACTER SET ANY_CS,
  len IN PLS_INTEGER DEFAULT NULL);

```

Если параметр `len` содержит `NULL`, Oracle заполняет буфер до максимального размера вплоть до достижения конца страницы, после чего операция чтения выдает исключение `UTL_HTTP.end_of_body` (да, это противоречит нормальной практике программирования, в соответствии с которой обычные операции не должны инициировать исключения). При каждой итерации обрабатывается содержимое очередного прочитанного фрагмента — например, оно присоединяется к объекту `LOB`.

Данные также можно выбирать построчно — для этого вместо `READ_TEXT` используется процедура `READ_LINE`:

```

PROCEDURE UTL_HTTP.read_line(
  r IN OUT NOCOPY UTL_HTTP.resp,
  data OUT NOCOPY VARCHAR2 CHARACTER SET ANY_CS,
  remove_crlf IN BOOLEAN DEFAULT FALSE);

```

Эта встроенная процедура читает одну строку исходного текста (по желанию программиста — с отсечением символов-завершителей). Недостаток `READ_LINE` заключается в том, что каждая строка, прочитанная с сервера HTTP, должна иметь длину менее 32 767 байт. Используйте процедуру `READ_LINE` только в том случае, если вы твердо уверены, что это ограничение не создаст проблем.

Загрузка страницы в объект LOB

Чтение данных по фрагментам или по строкам может привести к различным конфликтам ограничения длины, поэтому чтение данных в объекты `LOB` выглядит более логично. И снова Oracle предоставляет очень простую процедуру, которой во многих случаях оказывается достаточно. Вся страница загружается в простую структуру данных при помощи встроенного объектного типа `HTTPURITYPE`:

```

DECLARE
  text CLOB;
BEGIN
  text := HTTPURITYPE('http://www.oreilly.com').getclob;
END;

```

Загрузка двоичного файла в объект `BLOB` осуществляется методом `getblob()`:

```

DECLARE
  image BLOB;
BEGIN
  image :=
    HTTPURITYPE('www.oreilly.com/catalog/covers/oraclep4.s.gif').getblob;
END;

```

Конструктор HTTPURITYPE предполагает, что при передаче данных используется транспортный протокол HTTP, поэтому префикс «http://» можно не указывать. К сожалению, эта встроенная функция не поддерживает HTTPS и не позволяет передавать пользовательские заголовки «User-Agent».

Загрузка данных в LOB средствами UTL_HTTP осуществляется так:

```
/* Файл в Сети: url_to_clob.sql */
DECLARE
    req UTL_HTTP.req;
    resp UTL_HTTP.resp;
    buf VARCHAR2(32767);
    pagelob CLOB;
BEGIN
    req := UTL_HTTP.begin_request('http://www.oreilly.com/',
        http_version => UTL_HTTP.http_version_1_1);
    UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0 (compatible;
MSIE 6.0; Windows NT 5.1)');
    resp := UTL_HTTP.get_response(req);
    DBMS_LOB.createtemporary(pagelob, TRUE);
    BEGIN
        LOOP
            UTL_HTTP.read_text(resp, buf);
            DBMS_LOB.writeappend(pagelob, LENGTH(buf), buf);
        END LOOP;
    EXCEPTION
        WHEN UTL_HTTP.end_of_body
        THEN
            NULL;
    END;
    UTL_HTTP.end_response(resp);

    ...Код разбора данных, сохранения или иной обработки LOB

    DBMS_LOB.freetemporary(pagelob);
END;
```

Аутентификация HTTP

Хотя многие сайты (например, Amazon или Ebay) выполняют вход и аутентификацию с применением нестандартных форм HTML, остается еще немало сайтов, использующих аутентификацию HTTP, также называемую базовой аутентификацией. Такие сайты легко узнать по поведению браузера, в котором открывается модальное диалоговое окно с предложением ввести имя пользователя и пароль.

Иногда ввод данных в окне удастся обойти — для этого имя пользователя и пароль включаются в URL-адрес в следующей форме (хотя официальные стандарты не рекомендуют использовать этот метод):

```
http://username:password@some.site.com
```

Этот синтаксис поддерживается как пакетом UTL_HTTP, так и объектным типом HTTPURITYPE (по крайней мере с версии 9.2.0.4). Простой пример:

```
DECLARE
    webtext clob;
    user_pass VARCHAR2(64) := 'bob:swordfish'; -- замените своими данными
    url VARCHAR2(128) := 'www.encryptedsite.com/cgi-bin/login';
BEGIN
    webtext := HTTPURITYPE(user_pass || '@' || url).getclob;
END;
/
```

Если шифрование имени пользователя и пароля в URL-адресе не работает, можно попробовать другой способ:

```
...
req := UTL_HTTP.begin_request('http://some.site.com/');
UTL_HTTP.set_authentication(req, 'bob', 'swordfish');
resp := UTL_HTTP.get_response(req);
...
```

Он сработает в том случае, если сайт не шифрует страницу входа.

Загрузка зашифрованной страницы (HTTPS)

Хотя тип HTTPURITYPE не поддерживает загрузку данных через протокол SSL, UTL_HTTP решит эту задачу при наличии электронного бумажника Oracle — файла, содержащего сертификаты безопасности (а также, возможно, пары открытых и закрытых ключей). Для загрузки данных HTTPS понадобится первое, то есть сертификаты. Вы можете сохранить один или несколько электронных бумажников в файловой системе сервера базы данных или в службе каталогов LDAP. За дополнительной информацией об электронных бумажниках и других средствах безопасности Oracle рассказано в главе 23.

Чтобы создать электронный бумажник, необходимо запустить графическую программу Oracle под названием Oracle Wallet Manager; на компьютерах Unix/ Linux она обычно называется *owm*, а в Microsoft Windows вызывается из меню Пуск ► Oracle. Создав бумажник, попробуйте выполнить¹ следующий фрагмент:

```
DECLARE
    req UTL_HTTP.req;
    resp UTL_HTTP.resp;
BEGIN
    UTL_HTTP.set_wallet('file:/oracle/wallets', 'password1');
    req := UTL_HTTP.begin_request('https://www.entrust.com/');
    UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
    resp := UTL_HTTP.get_response(req);
    UTL_HTTP.end_response(resp);
END;
```

Если вы не получите сообщение об ошибке, радуйтесь — такое решение должно работать, потому что Entrust относится к числу немногих организаций, сертификаты которых включаются по умолчанию при создании электронного бумажника.

Если вы хотите загрузить данные из другого узла HTTPS, открытый сертификат которого не входит в список Oracle, снова запустите Oracle Wallet Manager, импортируйте сертификат в файл и снова разместите его на сервере. Чтобы загрузить сертификат в формате, пригодном к использованию, запустите Microsoft Internet Explorer и выполните следующие действия:

1. Откройте в браузере (Microsoft IE) сайт HTTPS.
2. Сделайте двойной щелчок на желтом значке с изображением замка в правом нижнем углу окна.
3. Выберите команду Details ► Copy to File.
4. Выполните инструкции по экспортированию сертификата в кодировке base64.

Или, если на вашем компьютере установлен пакет OpenSSL (как правило, в системах Unix/Linux), выполните следующую команду:

```
echo '' | openssl s_client -connect хост:порт
```

¹ Спасибо Тому Кайту за доступное изложение этой темы в блоге (<http://asktom.oracle.com>).

Команда выдает обширную информацию в `stdout`; просто сохраните текст между строками `BEGIN CERTIFICATE` и `END CERTIFICATE` (включительно) в файле. Кстати говоря, HTTPS обычно использует порт 443.

При наличии сертификата вы можете выполнить следующие действия:

1. Откройте Oracle Wallet Manager.
2. Откройте файл электронного бумажника.
3. Импортируйте сертификат из только что созданного файла.
4. Сохраните файл электронного бумажника и отправьте его на сервер базы данных.

Помните, что сертификаты окажутся в электронном бумажнике Oracle только после их импортирования через Oracle Wallet Manager. Разумеется, электронный бумажник может содержать несколько сертификатов, а в каталоге могут храниться несколько электронных бумажников.

Передача данных методами GET и POST

Еще одна распространенная задача — получение данных от сайта так, словно вы заполнили форму в браузере и нажали кнопку `Submit`. В этом разделе приведены некоторые примеры использования `UTL_HTTP` для решения этой задачи, но многие сайты весьма капризны, и чтобы передача данных работала правильно, придется изрядно потрудиться. Некоторые навыки и инструменты, которые пригодятся вам в ходе анализа поведения сайтов:

- Хорошее знание исходного кода HTML (особенно в области форм HTML) и, возможно, JavaScript.
- Режим просмотра исходного кода в браузере.
- Программы типа GNU *wget*, позволяющие легко опробовать разные URL-адреса с возможностью просмотра обычно скрытых взаимодействий между клиентом и сервером (ключ `-d`).
- Подключаемые модули для браузера — такие, как Web Developer Криса Педерика (Chris Pederick) или Tamper Data Адама Джадсона (Adam Judson) для браузеров на базе Mozilla.

Начнем с простого кода, который может использоваться для запросов к Google.

На главной странице Google используется одна форма HTML:

```
<form action=/search name=f>
```

Поскольку тег `method` отсутствует, по умолчанию используется метод GET. Единственное текстовое поле `q` на форме обладает следующими свойствами (среди прочих):

```
<input autocomplete="off" maxLength=2048 size=55 name=q value="">
```

Запрос GET можно закодировать непосредственно в URL-адресе:

```
http://www.google.com/search?q=query
```

Располагая такой информацией, мы можем написать следующий программный эквивалент поиска строки `"oracle pl/sql programming"` (включая двойные кавычки) в Google:

```
DECLARE
    url VARCHAR2(64)
        := 'http://www.google.com/search?q=';
    qry VARCHAR2(128) := UTL_URL.escape('"oracle pl/sql programming"', TRUE);
    result CLOB;
BEGIN
    result := HTTPURITYPE(url || qry).getclob;
END;
```

Удобная функция Oracle `UTL_URL.ESCAPE` заменяет специальные символы в строке запроса их шестнадцатеричными эквивалентами. Преобразованный текст в данном примере принимает следующий вид:

```
%22oracle%20pl%2Fsql%20programming%22
```

Давайте рассмотрим пример использования POST в чуть менее тривиальной ситуации. Анализ исходного кода HTML сайта <http://www.apache.org> показал, что в качестве «действия» формы поиска задано обращение к <http://search.apache.org>, что форма использует метод POST, а полю поиска присвоено имя `query`. При использовании метода POST данные не удастся присоединить к URL; вместо этого они должны пересылаться веб-серверу в определенной форме. Следующий блок выдает запрос POST на поиск строки Oracle `pl/sql` (основные изменения выделены жирным шрифтом):

```
DECLARE
    req UTL_HTTP.req;
    resp UTL_HTTP.resp;
    qry VARCHAR2(512) := UTL_URL.escape('query=oracle pl/sql');
BEGIN
    req := UTL_HTTP.begin_request('http://search.apache.org/', 'POST', 'HTTP
/1.0');
    UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
    UTL_HTTP.set_header(req, 'Host', 'search.apache.org');
    UTL_HTTP.set_header(req, 'Content-Type', 'application/x-www-form-urlencoded');
    UTL_HTTP.set_header(req, 'Content-Length', TO_CHAR(LENGTH(qry)));
    UTL_HTTP.write_text(req, qry);
    resp := UTL_HTTP.get_response(req);
```

...Построчная обработка результатов

```
    UTL_HTTP.end_response(resp);
END;
```

В двух словах: в `BEGIN_REQUEST` включается директива POST, а метод `write_text` используется для передачи данных формы. Хотя POST не позволяет присоединять пары «имя/значение» в конец URL-адреса (в отличие от запросов GET), данный сайт поддерживает тип контента `x-www-form-urlencoded` с включением пар «имя/значение» в отправляемую серверу переменную `qry`.

В этом примере встречается один дополнительный заголовок, который не используется в других примерах:

```
UTL_HTTP.set_header(req, 'Host', 'search.apache.org');
```

Без этого заголовка сайт Apache выдает свою главную страницу вместо поисковой. Заголовок «Host» необходим для сайтов с «виртуальными хостами» (то есть двумя и более именами хостов, представленным одним IP-адресом), чтобы веб-сервер знал, какую информацию вы запрашиваете. К счастью, передача заголовка «Host» не создает никакого риска даже в том случае, если удаленный сайт не поддерживает виртуальные хосты.

Кстати говоря, если форма содержит несколько заполняемых элементов, по правилам кодирования URL пары «имя/значение» должны разделяться символами `&`:

```
name1=value1&name2=value2&name3= ...
```

Итак, GET и POST успешно работают, можно переходить к получению данных... верно? Возможно. Скорее всего, ваш код рано или поздно столкнется с механизмом «перенаправления» HTTP. Это специальный код, который возвращается веб-сервером и означает: «Извините, но для этого вам нужно перейти в другое место». Все мы привыкли, что браузеры выполняют перенаправление автоматически и без лишних хлопот, однако на самом деле реализация может быть весьма непростой: существует минимум пять разных

видов перенаправления, для которых действуют несколько различающиеся правила относительно того, что «можно» делать браузеру. Перенаправление может встретиться на любой веб-странице, но во многих случаях можно воспользоваться функциональностью отслеживания перенаправлений UTL_HTTP:

UTL_HTTP.set_follow_redirect (максимум IN PLS_INTEGER DEFAULT 3);

К сожалению, в процессе тестирования кода для загрузки страницы с прогнозом погоды с сайта Национальной метеорологической службы США я обнаружил, что сервер реагирует на POST кодом 302 — странный особый случай в стандарте HTTP, который означает, что клиент *не должен* следовать за перенаправлением... и пакет UTL_HTTP соблюдает «букву» стандарта — по крайней мере в этом случае.

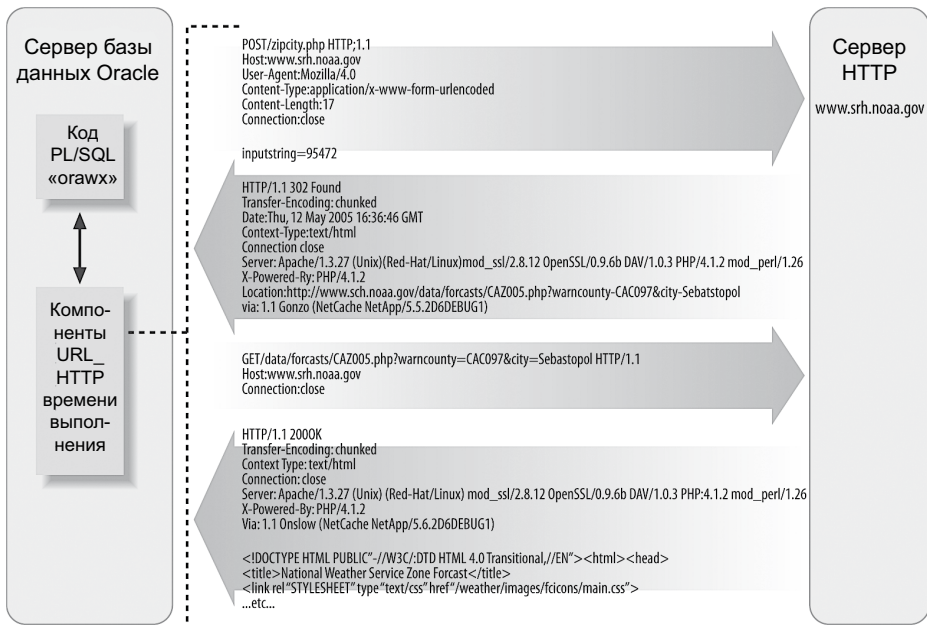


Рис. 22.2. Получение сводки погоды с сайта NOAA требует

Итак, чтобы получить полезную информацию о погоде, пришлось проигнорировать стандарт. Итоговая версия моей программы для получения погодной сводки в Севастополе (штат Калифорния) выглядит так:

```
/* Файл в Сети: orawx.sp */
PROCEDURE orawx
AS
    req UTL_HTTP.req;
    resp UTL_HTTP.resp;
    line VARCHAR2(32767);
    formdata VARCHAR2(512) := 'inputstring=95472'; -- zip code
    newlocation VARCHAR2(1024);
BEGIN
    req := UTL_HTTP.begin_request('http://www.srh.noaa.gov/zipcity.php',
        'POST', UTL_HTTP.http_version_1_0);
    UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
    UTL_HTTP.set_header(req, 'Content-Type', 'application/x-www-form-urlencoded');
    UTL_HTTP.set_header(req, 'Content-Length', TO_CHAR(LENGTH(formdata)));
    UTL_HTTP.write_text(req, formdata);
    resp := UTL_HTTP.get_response(req);
```



```

IF resp.status_code = UTL_HTTP.http_found
THEN
  UTL_HTTP.get_header_by_name(resp, 'Location', newlocation);
  req := UTL_HTTP.begin_request(newlocation);
  resp := UTL_HTTP.get_response(req);    END IF;

```

...Обработка страницы, как и прежде...

```

  UTL_HTTP.end_response(resp);
END;

```

На рис. 22.2 представлена схема взаимодействий между кодом и сервером. Не знаю, насколько часто встречается эта проблема. Моя «заплатка» не является универсальным решением для любых перенаправлений, но она дает представление о странностях, с которыми вы можете столкнуться при написании такого кода.

Запрет и долгосрочное хранение cookie

Плохо ли, хорошо ли, но поддержка cookie на уровне сеанса включена по умолчанию в последних версиях UTL_HTTP. Oracle по умолчанию разрешает до 20 cookie на сайт и до 300 cookie на сеанс. Чтобы проверить, относятся ли эти ограничения к вашей версии UTL_HTTP, используйте следующий фрагмент:

```

DECLARE
  enabled BOOLEAN;
  max_total PLS_INTEGER;
  max_per_site PLS_INTEGER;
BEGIN
  UTL_HTTP.get_cookie_support(enabled, max_total, max_per_site);
  IF enabled
  THEN
    DBMS_OUTPUT.PUT('Allowing ' || max_per_site || ' per site');
    DBMS_OUTPUT.PUT_LINE(' for total of ' || max_total || ' cookies. ');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Cookie support currently disabled. ');
  END IF;
END;

```

Поддержка cookie прозрачна; Oracle автоматически сохраняет cookie в памяти и отправляет их серверу по запросу.

Cookie исчезают при завершении сеанса. Если вы предпочитаете продлить срок их существования, сохраните их в таблицах Oracle и восстановите в начале нового сеанса. Для этого обратитесь к примерам кода, предоставляемого Oracle в разделе UTL_HTTP руководства *Packages and Types*.

Чтобы полностью отключить поддержку cookie для всех запросов UTL_HTTP для всех последующих запросов сеанса, используйте следующий код:

```
UTL_HTTP.set_cookie_support (FALSE);
```

Запрет cookie для конкретного запроса реализуется следующим образом:

```
UTL_HTTP.set_cookie_support(req, FALSE);
```

Чтобы изменить количество cookie в значениях по умолчанию, выполните команду:

```

UTL_HTTP.set_cookie_support(TRUE,
  max_cookies => n,
  max_cookies_per_site => m);

```

Загрузка данных с сервера FTP

Oracle не предоставляет встроенной поддержки загрузки данных с сайтов FTP из кода PL/SQL. Впрочем, если вам потребуется отправить или загрузить файлы по протоколу

FTP, в Интернете можно найти несколько готовых решений. Я видел как минимум три разных пакета, авторами которых были Барри Чейз (Barry Chase), Тим Холл (Tim Hall) и Крис Пул (Chris Poole). Эти реализации обычно используют пакеты UTL_TCP и UTL_FILE (и возможно, Java) и поддерживают большую часть стандартных операций FTP. Ссылки на некоторые реализации размещены на сайте PLNet.org.

Кроме того, некоторые прокси-серверы поддерживают загрузку FTP на базе запросов HTTP от клиента; возможно, это позволит вам обойтись без полноценного пакета FTP.

Использование прокси-сервера

По разным причинам в корпоративных сетях весь трафик часто передается через прокси-сервер. К счастью, в Oracle такой сценарий предусмотрен в UTL_HTTP. Например, если прокси-сервер использует порт 8888 по адресу 10.2.1.250, вы можете использовать следующий код:

```
DECLARE
    req UTL_HTTP.req;
    resp UTL_HTTP.resp;
BEGIN
    UTL_HTTP.set_proxy(proxy => '10.2.1.250:8888',
                      no_proxy_domains => 'mycompany.com, hr.mycompany.com');

    req := UTL_HTTP.begin_request('http://some-remote-site.com');

    /* Если прокси-сервер требует аутентификации, используйте: */
    UTL_HTTP.set_authentication(r => req,
                               username => 'имя_пользователя',
                               password => 'пароль',
                               for_proxy => TRUE);
    resp := UTL_HTTP.get_response(req);... etc.
```

Я тестировал этот код на прокси-сервере, использующем аутентификацию на базе Microsoft NTLM. После многочисленных проб и ошибок выяснилось, что перед именем пользователя следует поставить префикс из доменного имени сервера Microsoft и обратной косой черты. Другими словами, если обычно я подключаюсь к домену NTLM «mis» как пользователь bill с паролем swordfish, параметры должны выглядеть так:

```
username => 'mis\bill', password => 'swordfish'
```

Другие разновидности ввода/вывода в PL/SQL

В этой главе были кратко представлены некоторые механизмы ввода/вывода, особенно часто встречающиеся в реальных приложениях, однако существуют и другие:

- Каналы (pipes) баз данных, очереди и оповещения.
- Сокеты TCP.
- Встроенный веб-сервер Oracle.

Каналы, очереди и оповещения

Встроенный пакет DBMS_PIPE изначально проектировался как эффективный механизм передачи небольших блоков данных между сеансами Oracle. С появлением автономных транзакций потребность в каналах как механизме простой изоляции транзакций друг от друга отпала. Кроме того, каналы могут использоваться для самостоятельной организации параллельных операций.

Очереди — другой механизм асинхронной передачи сообщений между сеансами Oracle. Они существуют в нескольких разновидностях: с одним и несколькими производителями данных (producers), с одним и несколькими потребителями данных (consumers), с ограничением срока жизни сообщений, с приоритетами и т. д. Новейшая функциональность очередей в Oracle описана в документации Oracle *Streams Advanced Queuing User's Guide*.

Пакет DBMS_ALERT обеспечивает синхронное оповещение нескольких сеансов о наступлении различных событий базы данных. В наши дни эта возможность используется относительно редко; Oracle предоставляет другие средства, которые решают те же задачи, но с более мощной функциональностью.

Сокеты TCP

Какой бы интересной ни считалась тема низкоуровневого сетевого программирования среди специалистов, она просто не относится к числу часто используемых возможностей. В дополнение к встроенному пакету UTL_TCP Oracle также поддерживает вызов сетевых функций из хранимых процедур Java, которые могут активизироваться из кода PL/SQL.

Встроенный веб-сервер Oracle

Даже если вы не приобрели лицензию на продукт Oracle Application Server, это не мешает вам использовать сервер HTTP, встроенный в Oracle. Конфигурация встроенного сервера изменяется в зависимости от версии Oracle, но программирование на PL/SQL для него (включая пакеты OWA_UTIL, HTTP и HTTPF) остается относительно неизменным.

Эти пакеты позволяют генерировать веб-страницы на основании данных из базы непосредственно в коде PL/SQL. Данная тема весьма обширна, особенно если вы собираетесь генерировать и обрабатывать формы HTML на своей странице. В книге *Learning Oracle PL/SQL* (издательство O'Reilly) представлено введение в PL/SQL с интенсивным использованием встроенного веб-сервера и многочисленными примерами. Приемы программирования PL/SQL также применимы при использовании автономного полноценного сервера приложений Oracle; за дополнительной информацией об этом продукте обращайтесь к книге *Oracle Application Server 10g Essentials* (авторы — Рик Гринуолд (Rick Greenwald), Роберт Стаковяк (Robert Stackowiak) и Дональд Бейлз (Donald Bales)).

Хотя Oracle Application Express (или Oracle APEX) формально не является механизмом ввода/вывода, о нем также стоит упомянуть. Эта бесплатная надстройка для Oracle позволяет строить полнофункциональные веб-приложения, подключающиеся к базе данных Oracle. Программисты PL/SQL могут писать собственные хранимые программы, которые интегрируются с графической инфраструктурой Oracle APEX, предоставляющей много удобных средств для обмена данными через наглядный пользовательский интерфейс.

VI

Особые возможности PL/SQL

В арсенале такого мощного и богатого языка, как PL/SQL, имеется множество элементов, которыми программисты пользуются сравнительно редко, но обойтись без которых в особо сложных и ответственных ситуациях просто невозможно. В этой части книги затрагиваются наиболее сложные аспекты в изучении компонентов и возможностей PL/SQL. В главе 23 рассматриваются проблемы безопасности, с которыми мы сталкиваемся при построении программ PL/SQL. Глава 24 содержит описание архитектуры PL/SQL, в том числе использования памяти. Глава 25 предназначена для разработчиков PL/SQL, которым приходится сталкиваться с проблемами глобализации и локализации. Наконец, глава 26 содержит введение в объектно-ориентированные средства Oracle.

23

Безопасность и PL/SQL

Многие разработчики PL/SQL считают, что о безопасности должны заботиться только администраторы баз данных и специалисты по безопасности. Действительно, некоторыми аспектами безопасности (например, управлением пользователями и привилегиями) должны заниматься администраторы. Но было бы серьезной ошибкой полагать, что разработчиков PL/SQL эта тема вообще не касается. Во-первых, безопасность не является проблемой, которая решается один раз; это средство достижения цели, о котором приходится помнить постоянно. Во-вторых, многие администраторы направляют свои усилия на защиты базы данных в целом, а не на программирование средств безопасности конкретного приложения.

Вероятно, вы слышали, что надежность системы определяется надежностью ее слабейшего звена. Этот принцип в равной степени применим к безопасности приложений. Каждый элемент инфраструктуры — приложение, архитектура, промежуточные программы (middleware), база данных, операционная система — вносит свой вклад в общую безопасность инфраструктуры, и сбой безопасности одного компонента создает угрозу для всей системы. Понимание структурных элементов безопасности и внедрение их в архитектуру приложения не только желательно, но и абсолютно необходимо.

Общие сведения о безопасности

Аспекты безопасности Oracle делятся на три общие категории:

- Находящиеся исключительно в ведении администраторов баз данных, систем и сетей. Эти вопросы (например, управление пользователями и привилегиями) выходят за рамки книги.
- Аспекты, важные для разработчиков и проектировщиков архитектуры, не относящиеся к сфере ответственности администраторов. Одним из примеров служит выбор модели прав вызова при создании хранимого кода; как правило, решение принимается в ходе проектирования самим разработчиком, а не администратором. Эти вопросы рассматриваются в других главах книги (например, модели прав вызова описаны в главе 24).
- Аспекты, которые обычно считаются относящимися к сфере администрирования, но которые должны быть хорошо известны разработчикам и проектировщикам. К этой категории относится шифрование, безопасность уровня строк и контексты приложений. Именно этим аспектам посвящена эта глава.

Как возможности и средства, описанные в этой главе, помогут разработчикам PL/SQL и проектировщикам архитектуры приложений в их работе? Чтобы ответить на этот вопрос, следует рассмотреть каждую тему по отдельности:

- **Шифрование** играет жизненно важную роль для защиты данных и активно применяется во многих ситуациях при проектировании приложений. Вы должны обладать практическими знаниями в области функциональности Oracle, относящейся к шифрованию, включая механизм прозрачного шифрования данных (TDE, Transparent Data Encryption), представленный в Oracle Database 10g Release 2, и механизм прозрачного шифрования таблиц (TTE, Transparent Table Encryption), представленный в Oracle Database 11g.
- **Безопасность уровня строк (RLS, Row Level Security)**. При проектировании приложения необходимо хорошо понимать архитектуру доступа и проверки полномочий для работы с данными. RLS позволяет ограничить строки, доступные для пользователя. Во многих случаях RLS упрощает понимание и реализацию приложений, а иногда даже автоматически обеспечивает соответствие готовых приложений правилам безопасности, принятым в вашей организации.
- **Контексты приложений** представляют собой пары «имя/значение», определяемые в сеансе посредством выполнения специальных хранимых процедур. Контексты приложений обычно используются для управления доступом к ресурсам базы данных в соответствии с правилами, изменяющимися в зависимости от текущего пользователя.
- **Детализированный аудит FGA (Fine-Grained Auditing)** предоставляет механизм для регистрации факта выдачи пользователями определенных команд и выполнения некоторых условий. FGA предоставляет в распоряжение разработчика немало полезных функций. Например, FGA фактически позволяет реализовать триггеры SELECT — пользовательские процедуры, которые должны автоматически выполняться при каждой выборке данных из таблицы.

Тема безопасности в Oracle воистину необъятна; в этой главе мы лишь в самых общих чертах рассмотрим ее аспекты, представляющие наибольший интерес для разработчиков PL/SQL. За дополнительной информацией об этих и других операциях безопасности Oracle обращайтесь к книге *Oracle PL/SQL for DBAs* (Arup Nanda, Steven Feuerstein, O'Reilly). Также выпущено немало других книг, которые помогут вам разобраться в тонкостях вопросов безопасности, затронутых в этой главе. Ресурсы по этой теме также доступны на сайте Oracle Security Technology Center (<http://www.oracle.com/technetwork/security>).

Шифрование

В простейшем определении *шифрование* представляет собой «маскировку» данных, или их преобразование, при котором данные не могут использоваться посторонними. Рассмотрим очень простой пример: я ежедневно снимаю деньги со своего счета при помощи кредитной карты. Каждый раз я должен ввести в банкомате свой PIN-код. К сожалению, я от природы весьма забывчив, поэтому я решаю записать PIN-код на предмете, который всегда под рукой при использовании карты, — то есть на самой карте. При этом я отлично понимаю, что PIN-код, написанный на карте, существенно ослабляет защиту; если карту украдут, то похититель сразу увидит написанный на ней код. Прощайте, сбережения! Что делать, чтобы вор не смог узнать PIN-код, но при этом не забыть его самому?

Через пару минут меня озаряет гениальная идея: цифры надо изменить заранее определенным образом. Я прибавляю к PIN-коду число из одной цифры и записываю результат

на карте. Допустим, к PIN-коду 6523 прибавляется число 6; в сумме получается 6529, и я записываю это число на карте. Если карту украдут, то вор увидит код 6529, который будет совершенно бесполезен, потому что для восстановления PIN-кода нужно знать способ изменения исходного числа. Даже если он будет знать, что результат был получен сложением, ему нужно будет угадать число (6 в нашем случае). Иначе говоря, я зашифровал PIN-код и затруднил определение фактического значения.

Давайте ненадолго задержимся и подробнее рассмотрим механику, прежде чем я вернусь к своему примеру и признаюсь, что на самом деле идея не такая уж гениальная. Для проведения шифрования (то есть для того, чтобы исказить PIN-код до неузнаваемости) нужно знать две вещи:

- Метод изменения исходных данных (в данном случае прибавление числа к исходному значению).
- Конкретное прибавленное число (6).

Первая часть называется *алгоритмом*, а вторая — *ключом*. Это основные компоненты любой системы шифрования, схема которой изображена на рис. 23.1. Вы можете сохранять один компонент неизменным и изменять другой для получения другого экземпляра зашифрованных данных.

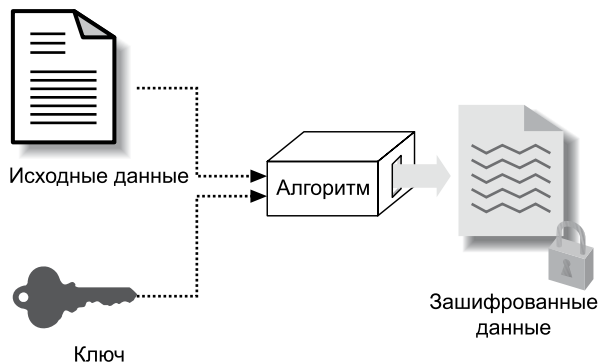


Рис. 23.1. Основная схема шифрования

В действительно безопасных системах используются чрезвычайно сложные алгоритмы. Вообще говоря, для построения системы шифрования вам не обязательно хорошо разбираться в них, однако вы должны знать основные типы алгоритмов, их сильные и слабые стороны. Впрочем, один лишь выбор алгоритма еще не гарантирует безопасности. Она обеспечивается изменением другой переменной, находящейся под вашим ведением, — *ключа шифрования*.

Одной из важнейших проблем при построении инфраструктуры шифрования является построение эффективной системы управления ключами. Если злоумышленник получит доступ к ключам шифрования, то зашифрованные данные окажутся под угрозой независимо от сложности алгоритма. С другой стороны, некоторые пользователи (например, приложения) должны иметь доступ к ключам для своей работы, и он должен быть достаточно простым для нормальной работы приложения. Проблема заключается в определении оптимального соотношения между простотой доступа и безопасностью ключей. Позднее в этой главе будет рассмотрен пример создания эффективной системы управления ключами.

Длина ключа

У приведенного выше примера с шифрованием PIN-кода имеется серьезный недостаток. Вор может догадаться, что я просто прибавил число к PIN-коду, чтобы зашифровать его. Конечно, он не знает, что это за число, но он может перебирать возможные значения. Если я использую число из одной цифры, ему понадобится не более 10 попыток. Но предположим, я использую число из двух цифр — теперь придется угадывать число от 0 до 99, а число попыток увеличивается до 100. Увеличение количества цифр в ключе усложняет «взлом» шифра. Таким образом, длина ключа играет исключительно важную роль в повышении безопасности любой системы шифрования.

Конечно, в реальных схемах шифрования длина ключа не ограничивается одной-двумя цифрами. Более того, ключи вообще не состоят из одних цифр. Их длина обычно составляет не менее 56 бит, но может достигать и до 256 бит. Длина ключа зависит от выбора алгоритма (см. следующий раздел).



Чем длиннее ключ, тем сложнее взломать шифр. С другой стороны, длинные ключи увеличивают затраты времени на шифрование и дешифрование, потому что процессору приходится выполнять больший объем работы.

Алгоритмы

Существует немало распространенных, пользующихся коммерческой поддержкой алгоритмов шифрования. Впрочем, нас интересуют алгоритмы, поддерживаемые Oracle для приложений PL/SQL. Все эти алгоритмы относятся к категории алгоритмов с закрытым ключом (иногда называемых *симметричными* алгоритмами); основные их отличия от алгоритмов с открытым ключом (иногда называемых *асимметричными*) описаны во врезке.

В Oracle чаще всего используются следующие алгоритмы:

- **DES** (Data Encryption Standard). Традиционно алгоритм DES занимал ведущие позиции в области шифрования. Он был разработан более 20 лет назад для Национального бюро стандартов (позднее переименованного в Национальный институт стандартов и технологий), и с тех пор был принят в качестве стандарта ISO. Об алгоритме DES и его истории можно рассказать очень много, но моей задачей является не описание алгоритма, а краткое описание его применения. Алгоритму DES необходим 64-разрядный ключ, но 8 бит ключа не используются. Чтобы подобрать ключ, злоумышленнику придется перебрать до 72 057 594 037 927 936 комбинаций. Возможностей DES было достаточно в течение нескольких десятилетий, но сейчас он постепенно уходит в прошлое. Современные мощные компьютеры способны перебрать даже огромное число комбинаций, необходимое для взлома ключа DES.
- **DES3**. В этой схеме, базирующейся на исходном алгоритме DES, данные шифруются дважды или трижды (в зависимости от режима вызова). DES3 использует 128- или 192-разрядный ключ; его длина определяется количеством проходов. Надежность алгоритма DES3 тоже была приемлемой в течение некоторого времени, но сейчас и этот алгоритм постепенно устаревает и не обеспечивает защиты от целенаправленных атак.
- **AES**. В ноябре 2001 года был одобрен новый стандарт AES (Advanced Encryption Standard), вступивший в силу в мае 2002 года. Полный текст стандарта можно найти по адресу <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

ОТКРЫТЫЙ ИЛИ ЗАКРЫТЫЙ КЛЮЧ?

В схемах шифрования с закрытым ключом (симметричных) один ключ сначала используется для шифрования данных, а затем для их дешифрования. Для дешифрования необходимо иметь доступ к ключу, который должен быть каким-то образом передан пользователю. Это может создать неудобства, так как передача ключа усложняет схемы управления ключами и снижает их надежность.

С другой стороны, при шифровании с открытым ключом (асимметричным) предполагаемый получатель данных генерирует два ключа. Один ключ (закрытый) остается у него, а другой (открытый) передается предполагаемому отправителю. Последний шифрует данные с использованием открытого ключа, но эти данные могут быть расшифрованы только с закрытым ключом. Отправитель не знает закрытого ключа и не может внести какие-либо модификации в передаваемые данные. Открытые ключи выдаются до пересылки данных и могут использоваться многократно. В такой схеме управление ключами чрезвычайно упрощается, что решает как минимум одну проблему шифрования.

Открытые и закрытые ключи статистически связаны, поэтому теоретически закрытый ключ может быть восстановлен по открытому ключу, хотя это потребует огромного объема работы. Для снижения риска подбора ключей методом «грубой силы» в асимметричных схемах используются очень длинные ключи (обычно 1024-разрядные), в отличие от 64-, 128- и 256-разрядных ключей, используемых в симметричном шифровании.

Oracle предоставляет асимметричное шифрование в двух точках:

- При передаче данных между клиентом и базой данных.
- В процессе аутентификации пользователей.

Обе функции требуют применения Oracle Advanced Security Option — дополнения, отсутствующего по умолчанию. Программа просто включает асимметричное шифрование для этих функций; она не предоставляет инструментарий, который может использоваться разработчиками PL/SQL для шифрования хранимых данных.

Единственный инструмент Oracle, предназначенный для разработчиков, ориентирован на симметричное шифрование. По этой причине в данной главе асимметричное шифрование не рассматривается.

Заполнение и сцепление

Блок данных обычно не шифруется как единое целое. Чаще всего он разбивается на фрагменты по 8 байт, после чего каждый фрагмент шифруется независимо от других. Конечно, длина данных может быть не кратной 8 — в этом случае алгоритм добавляет символы в последний фрагмент до 8 байт. Этот процесс называется *заполнением* (padding). Если злоумышленник угадает, какие данные использовались для заполнения, это может упростить подбор ключа. Для безопасного заполнения следует использовать метод, реализованный в Oracle, называемый PKCS#5 (Public Key Cryptography System #5). Другие режимы (с заполнением нулями и вообще без заполнения) также будут представлены ниже.

Если данные делятся на фрагменты, также должен существовать способ объединения смежных фрагментов; этот процесс называется *сцеплением* (chaining). Безопасность системы шифрования также зависит от того, как происходит соединение и шифрование фрагментов (независимо или в сочетании со смежными фрагментами). Самым распространенным форматом сцепления является формат CBC (Cipher Block Chaining);

в Oracle он выбирается при помощи константы, определенной во встроенном пакете CHAIN_CBC. Также используются режимы сцепления Electronic Code Book (CHAIN_ECB), Cipher Feedback (CHAIN_CFB) и Output Feedback (CHAIN_OFB). Они тоже будут представлены позднее в этой главе.

Пакет DBMS_CRYPTO

Итак, мы познакомились с основными структурными элементами схем шифрования. Давайте посмотрим, как создать инфраструктуру шифрования в PL/SQL с использованием встроенного пакета Oracle DBMS_CRYPTO.



Пакет DBMS_CRYPTO появился в Oracle10g. В более ранних версиях пакет DBMS_OBFUSCATION_TOOLKIT предоставлял похожую (но не идентичную) функциональность. Старый пакет все еще остается доступным, но сейчас он считается устаревшим, и вместо него рекомендуется использовать новый пакет.

Вспомните, что для выполнения шифрования кроме входных данных необходимы еще четыре компонента:

- ключ шифрования;
- алгоритм шифрования;
- метод заполнения;
- метод сцепления.

Ключ шифрования предоставляете вы, а остальные компоненты предоставляет Oracle. Выбор осуществляется при помощи соответствующих констант пакета DBMS_CRYPTO.

Алгоритмы

В табл. 23.1 перечислены константы DBMS_CRYPTO, позволяющие выбрать конкретный алгоритм и длину ключа. Ссылки на эти константы должны задаваться в формате *имя_пакета.имя_константы* — например, DBMS_CRYPTO.ENCRYPT_DES для выбора алгоритма DES.

Таблица 23.1. Константы алгоритмов пакета DBMS_CRYPTO

| Константа | Фактическая длина ключа | Описание |
|-------------------|-------------------------|---|
| ENCRYPT_DES | 56 | DES (по аналогии с DBMS_OBFUSCATION_TOOLKIT) |
| ENCRYPT_3DES_2KEY | 112 | Модифицированный DES3; блок шифруется в три прохода с двумя ключами |
| ENCRYPT_3DES | 156 | DES3; блок шифруется в три прохода |
| ENCRYPT_AES128 | 128 | AES |
| ENCRYPT_AES192 | 192 | AES |
| ENCRYPT_AES256 | 256 | AES |
| ENCRYPT_RC4 | — | Единственный потоковый шифр, поддерживаемый пакетом (предназначен для шифрования потоковых, а не дискретных данных) |

Заполнение и сцепление

Режимы заполнения и сцепления задаются при помощи констант пакета DBMS_CRYPTO, перечисленных в табл. 23.2.

Таблица 23.2. Методы заполнения и сцепления пакета DBMS_CRYPTO

| Константа | Метод заполнения/сцепления |
|-----------|--|
| PAD_PKCS5 | Заполнение PKCS#5 |
| PAD_ZERO | Заполнение нулями |
| PAD_NONE | Заполнение не используется; метод выбирается для данных, длина которых кратна 8 байтам |
| CHAIN_CBC | Cipher Block (самый распространенный метод) |
| CHAIN_CFB | Cipher Feedback |
| CHAIN_ECB | Electronic Code Book |
| CHAIN_OFB | Output Feedback |

Вам почти не придется беспокоиться о выборе конкретного метода заполнения или сцепления; как правило, разные методы предлагают специфические возможности, редко необходимые при разработке типичных систем.

Чаще всего на практике выбирается метод заполнения PKCS#5 с методом сцепления CBC. В этой главе они будут использоваться во всех случаях, кроме тех, где обратное явно указано в тексте.

Шифрование данных

Начнем с очень простого примера шифрования строки «Confidential Data» функцией `DBMS_CRYPTO.ENCRYPT`. Функция получает четыре аргумента:

- `src` — исходные данные, подлежащие шифрованию (должны иметь тип данных `RAW`).
- `key` — ключ шифрования (также `RAW`). Длина ключа должна соответствовать выбранному алгоритму. Например, для алгоритма DES она должна быть не менее 64 бит.
- `typ` — определение трех компонентов (алгоритм, механизм заполнения и метод сцепления) в виде суммы соответствующих констант.
- `iv` — необязательный вектор инициализации (IV), еще один компонент схемы шифрования, затрудняющий анализ «закономерностей» в зашифрованном тексте (эта тема выходит за рамки настоящей главы).

В следующих примерах будут использоваться:

- алгоритм — AES с 128-разрядным ключом;
- метод сцепления — CBC;
- механизм заполнения — PKCS#5.

Они задаются следующим значением параметра `typ` при вызове функции:

```
DBMS_CRYPTO.ENCRYPT_AES128
+ DBMS_CRYPTO.CHAIN_CBC
+ DBMS_CRYPTO.PAD_PKCS5;
```

Если бы вместо PKCS#5 был выбран режим без заполнения, значение выглядело бы так:

```
DBMS_CRYPTO.ENCRYPT_AES128
+ DBMS_CRYPTO.CHAIN_CBC
+ DBMS_CRYPTO.PAD_NONE;
```

Аналогичным образом задается любая другая комбинация алгоритма и метода сцепления.

Затем необходимо выбрать ключ. Предположим, в качестве ключа будет использоваться строка «1234567890123456». Это значение относится к типу данных `VARCHAR2`. Чтобы использовать его в функции `ENCRYPT`, необходимо сначала преобразовать его к типу `RAW`. Для этого мы воспользуемся функцией `STRING_TO_RAW` встроенного пакета `UTL_I18N` (этот пакет упоминается далее в настоящей главе):

```

DECLARE
    l_raw    RAW (200);
    l_in_val VARCHAR2 (200) := 'Confidential Data';
BEGIN
    l_raw := utl_i18n.string_to_raw (l_in_val, 'AL32UTF8');
END;
```

Переменная `l_in_val` типа `VARCHAR2` преобразована к типу `RAW`. Теперь можно переходить непосредственно к шифрованию входных данных:

```

/* Файл в Сети: enc.sql */
1  DECLARE
2      l_key    VARCHAR2 (2000) := '1234567890123456';
3      l_in_val VARCHAR2 (2000) := 'Confidential Data';
4      l_mod    NUMBER
5          :=  DBMS_CRYPTO.encrypt_aes128
6              + DBMS_CRYPTO.chain_cbc
7              + DBMS_CRYPTO.pad_pkcs5;
8      l_enc    RAW (2000);
9  BEGIN
10     l_enc :=
11         DBMS_CRYPTO.encrypt (utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
12                             l_mod,
13                             utl_i18n.string_to_raw (l_key, 'AL32UTF8')
14                             );
15     DBMS_OUTPUT.put_line ('Encrypted=' || l_enc);
16 END;
```

Результат:

Encrypted=C0777257DFBF8BA9A4C1F724F921C43C70D0C0A94E2950BBB6BA2FE78695A6FC

Давайте проанализируем этот код, строку за строкой.

| Строки | Описание |
|--------|--|
| 2 | Определение ключа. Длина ключа составляет ровно 16 символов (128 бит), в соответствии с требованиями AES. Если бы был выбран алгоритм AES192, я задал бы длину ключа равной $192/8=24$. При неверной длине ключа выдается исключение <code>KeyBadSize</code> |
| 3 | Входные данные для шифрования. Длина данных не ограничивается; допускается использование значения произвольной длины. Если длина не кратна 8 байтам, входные данные автоматически дополняются до нужной длины в соответствии с выбранным алгоритмом |
| 4–7 | Определение алгоритма, метода заполнения и метода сцепления |
| 8 | Определение переменной для хранения зашифрованного значения. Обратите внимание: выходные данные имеют тип <code>RAW</code> |
| 11 | Входные данные преобразуются из типа <code>VARCHAR2</code> в тип <code>RAW</code> |
| 13 | Ключ тоже должен передаваться функции в формате <code>RAW</code> |
| 15 | Вывод зашифрованного значения (тоже в формате <code>RAW</code>) в виде шестнадцатеричной строки. В реальной системе выводить значение бессмысленно; вероятно, с ним будет выполнена другая операция: сохранение в таблице, передача вызывающей процедуре для использования в другом месте и т. д. |

На базе `ENCRYPT` можно построить обобщенную функцию шифрования данных. В этой функции будет использоваться алгоритм AES с 128-разрядным ключом, метод заполнения `PKCS#5` и метод сцепления `CBC`. Таким образом, при вызове функции пользователь должен предоставить только шифруемые данные и ключ.

```

/* Файл в Сети: get_enc_eval.sql */
FUNCTION get_enc_val (p_in_val IN VARCHAR2, p_key IN VARCHAR2)
    RETURN VARCHAR2
IS
    l_enc_val    RAW (4000);
BEGIN
    l_enc_val :=
```

продолжение ➤

```

DBMS_CRYPTO.encrypt (src      => utl_i18n.string_to_raw (p_in_val,
                                                             'AL32UTF8'
                                                             ),
                    key       => utl_i18n.string_to_raw (p_key,
                                                             'AL32UTF8'
                                                             ),
                    typ       =>  DBMS_CRYPTO.encrypt_aes128
                               + DBMS_CRYPTO.chain_cbc
                               + DBMS_CRYPTO.pad_pkcs5
                    );

RETURN l_enc_val;
END;
```

Напоследок осталось упомянуть еще об одном обстоятельстве. Для преобразования данных `VARCHAR2` в `RAW` используется функция `UTL_I18N.STRING_TO_RAW` вместо `UTL_RAW.CAST_TO_RAW`. Почему?

Входные данные функции `ENCRYPT` должны иметь тип `RAW` и при этом использовать конкретный набор символов `AL32UTF8`, который может и не совпадать с набором символов базы данных. Следовательно, при проектировании строки `VARCHAR2` в `RAW` для шифрования необходимо выполнить два преобразования:

- Из текущего набора символов базы данных в набор `AL32UTF8`.
- Из `VARCHAR2` в `RAW`.

Оба преобразования выполняются функцией `STRING_TO_RAW` встроенного пакета `UTL_I18N`; функция `CAST_TO_RAW` не изменяет набор символов.



Пакет `UTL_I18N` является частью архитектуры Oracle Globalization Support и предназначен для глобализации (или интернализации) приложений. За дополнительной информацией о глобализации обращайтесь к главе 25.

Шифрование LOB

Большие объектные типы данных — такие, как `CLOB` и `BLOB`, — тоже могут шифроваться. Например, в данных `BLOB` могут храниться файлы сигнатур и копии юридических документов. Содержимое таких файлов конфиденциально, поэтому при хранении в базе данных их желательно зашифровать. Вместо того чтобы вызывать функцию `ENCRYPT`, как это делалось в предыдущих примерах, я воспользуюсь перегруженной процедурной версией `ENCRYPT`:

```

/* Файл в Сети: enc_lob.sql */
DECLARE
    l_enc_val  BLOB;
    l_in_val   CLOB;
    l_key      VARCHAR2 (16) := '1234567890123456';
BEGIN
    DBMS_CRYPTO.encrypt (dst      => l_enc_val,
                        src      => l_in_val,
                        key      => utl_i18n.string_to_raw (l_key, 'AL32UTF8'),
                        typ      =>  DBMS_CRYPTO.encrypt_aes128
                               + DBMS_CRYPTO.chain_cbc
                               + DBMS_CRYPTO.pad_pkcs5
                        );

END;
```

Результат сохраняется в переменной `l_enc_val`, которая затем может передаваться другим программам или сохраняться в таблице.



Для данных LOB используйте процедурную версию ENCRYPT; для всех остальных типов данных используйте функцию. Не забудьте преобразовать значения в формат RAW (а CLOB — в BLOB), прежде чем передавать их функции ENCRYPT.

SecureFiles

Большие объекты (LOB) были значительно переработаны в Oracle Database 11g; теперь для их обозначения используется термин *SecureFiles*. Традиционные объекты LOB (теперь называемые *BasicFiles*), такие как CLOB и BLOB, по-прежнему доступны, но я не рекомендую их использовать. В любых ситуациях, в которых в прошлом использовались LOB, теперь следует использовать SecureFiles. Технология SecureFiles предоставляет ту же функциональность, что и LOB, а также ряд дополнительных возможностей — таких, как сжатие, устранение дубликатов, кэширование, возможность прекращения ведения журнала и т. д. За дополнительной информацией об использовании SecureFiles обращайтесь к главе 13.

Дешифрование данных

Шифрование данных имеет смысл только в том случае, если зашифрованные данные в какой-то момент будут прочитаны и использованы в приложении. Эта задача решается при помощи функции DECRYPT. По структуре вызова она идентична функции ENCRYPT и получает те же четыре аргумента:

- `src` — зашифрованные данные.
- `key` — ключ, использованный для шифрования.
- `typ` — три компонента (алгоритм, механизм заполнения и метод сцепления), использованные при вызове ENCRYPT.
- `iv` — вектор инициализации, использованный при вызове ENCRYPT.

Функция DECRYPT тоже возвращает дешифрованные данные в формате RAW; для нормального просмотра их необходимо преобразовать в другой формат.

Давайте посмотрим, как работает дешифрование. В следующем примере зашифрованное значение сохраняется в переменной SQL*Plus, а затем используется в качестве исходных данных для функции DECRYPT.

```

1  /* Файл в decval.sql */
2  REM Определение переменной для хранения зашифрованного значения
3  VARIABLE enc_val varchar2(2000);
4  DECLARE
5      l_key          VARCHAR2 (2000) := '1234567890123456';
6      l_in_val       VARCHAR2 (2000) := 'Confidential Data';
7      l_mod          NUMBER
8          := DBMS_CRYPTO.encrypt_aes128
9             + DBMS_CRYPTO.chain_cbc
10            + DBMS_CRYPTO.pad_pkcs5;
11      l_enc          RAW (2000);
12  BEGIN
13      l_enc :=
14          DBMS_CRYPTO.encrypt (utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
15                              l_mod,
16                              utl_i18n.string_to_raw (l_key, 'AL32UTF8'))
17      DBMS_OUTPUT.put_line ('Encrypted=' || l_enc);
18      :enc_val := RAWTOHEX (l_enc);
19  END;
```

продолжение ➤

```

20 /
21 DECLARE
22     l_key          VARCHAR2 (2000) := '1234567890123456';
23     l_in_val       RAW (2000)      := HEXTORAW (:enc_val);
24     l_mod          NUMBER
25     :=            DBMS_CRYPTO.encrypt_aes128
26     +            DBMS_CRYPTO.chain_cbc
27     +            DBMS_CRYPTO.pad_pkcs5;
28     l_dec          RAW (2000);
29 BEGIN
30     l_dec :=
31         DBMS_CRYPTO.decrypt (l_in_val,
32                             l_mod,
33                             utl_i18n.string_to_raw (l_key, 'AL32UTF8')
34                             );
35     DBMS_OUTPUT.put_line ('Decrypted=' || utl_i18n.raw_to_char (l_dec));
36 END;
```

Основные моменты этого кода разъясняются в следующей таблице.

| Строки | Описание |
|--------|---|
| 22 | Объявление ключа для дешифрования. Обратите внимание: он должен совпадать с ключом, использованным при шифровании |
| 23 | Так как переменная enc_val содержит шестнадцатеричное значение, мы преобразуем ее к типу RAW |
| 25–27 | Как и при шифровании, необходимо задать алгоритм, методы заполнения и сцепления в одном параметре. Это значение должно совпадать с тем, которое использовалось при шифровании |
| 33 | Как и при шифровании, ключ должен храниться в переменной типа RAW, поэтому он преобразуется из VARCHAR2 в RAW |

Код выводит строку «Confidential Data» — ту, которая была изначально зашифрована.



Для расшифровки зашифрованного объекта LOB необходимо использовать перегруженную процедурную версию DECRYPT, потому что для шифрования использовалась процедурная версия ENCRYPT.

Генерирование ключей

До настоящего момента все наше внимание было сосредоточено на операциях шифрования и дешифрования, и в примерах использовался очень простой ключ «1234567890123456». Безопасность системы шифрования полностью зависит от безопасности ключа, то есть от тех трудностей, с которыми столкнется потенциальный злоумышленник при *подборе* значения ключа. Следовательно, ключ должен быть достаточно случайным, чтобы его нельзя было просто угадать.

В стандарте ANSI X9.31: Pseudo-Random Number Generator (PRNG) определяется стандартный алгоритм создания случайных чисел. Oracle реализует этот алгоритм в функции RANDOMBYTES пакета DBMS_CRYPTO. Функция получает один аргумент с длиной генерируемой случайной строки и возвращает значение заданной длины в формате RAW. Пример ее использования для создания 16-байтового значения:

```

DECLARE
    l_key  RAW (16);
BEGIN
    l_key := DBMS_CRYPTO.randombytes (16);
END;
```


Конечно, строка случайных байтов генерируется не просто так, а как ключ шифрования. При помощи этой функции можно сгенерировать ключ любой длины, соответствующей выбранному вами алгоритму.

Управление ключами

Мы рассмотрели азы шифрования/дешифрования, научились генерировать ключи. Но это была самая простая часть; в основном я просто показывал, как пользоваться готовыми средствами Oracle для достижения желаемой цели. Пора переходить к самой сложной части инфраструктуры шифрования — управлению ключами. Ключ должен быть доступен для наших приложений, чтобы они могли дешифровать зашифрованные значения, и механизм обращения к ключу должен быть по возможности простым. С другой стороны, поскольку ключ буквально является «ключом», защищающим зашифрованные данные, он не должен быть слишком доступным. Качественная схема управления ключами совмещает доступность ключей с предотвращением несанкционированного доступа к ним.

Известны три основных схемы управления ключами:

- Один ключ для всей базы данных.
- Один ключ для каждой строки таблицы с зашифрованными данными.
- Комбинация этих двух решений.

Ниже приводятся краткие описания этих разных подходов к управлению ключами.

Один ключ для всей базы данных

В этой схеме один ключ может использоваться для обращения к любым данным в базе. Как показано на рис. 23.2, процедура шифрования читает один ключ из того места, где он хранится, и шифрует все данные, нуждающиеся в защите.



Рис. 23.2. Схема с одним ключом

Ключ может храниться в разных местах:

- **В базе данных.** Самая простая стратегия из всех. Ключ хранится в реляционной таблице, возможно, в схеме, предназначенной специально для этой цели. Поскольку ключ хранится в базе данных, он автоматически сохраняется при создании резервной копии базы данных; старые значения ключа могут быть получены при помощи ретроспективных запросов, а сам ключ защищен от похищения на уровне операционной системы. Простота имеет и обратные стороны; так как ключ представляет собой обычные данные в таблице, любой пользователь с полномочиями модификации таблицы (например, администратор базы данных) может изменить ключ и разрушить всю инфраструктуру шифрования.
- **В файловой системе.** Ключ хранится в файле и читается процедурой шифрования с использованием встроенного пакета `UTL_FILE`. Задавая соответствующие привилегии доступа к файлу, можно защитить ключ от изменения из базы данных.
- **На съемном носителе под контролем пользователя.** Это самый безопасный способ — никто, кроме пользователя, не сможет расшифровать данные или изменить ключ, даже администратор базы данных или системы. В качестве съемного носителя может использоваться USB-диск, DVD или даже съемный жесткий диск. Главным недостатком съемного носителя является возможность потери ключа. Ответственность за безопасность ключа возлагается на пользователя. Если ключ будет потерян, то все зашифрованные данные пропадут — безвозвратно.

Главный недостаток этого способа — его зависимость от единой точки отказа. Если злоумышленник взломает базу данных и определит ключ, то вся база данных немедленно оказывается под его контролем. Кроме того, если вы захотите сменить ключ, вам придется изменять все строки всех таблиц; в большой базе данных объем работы может оказаться весьма значительным.

Один ключ для каждой строки

В этом варианте ключ связывается с каждой строкой таблицы, как показано на рис. 23.3. В базе данных создается отдельная таблица для хранения ключей. Исходная таблица и таблица ключей связываются через первичный ключ исходной таблицы.

Главное преимущество этого подхода заключается в том, что каждая запись защищается собственным ключом. Если один ключ будет подобран, то под контролем злоумышленника окажется только одна строка, а не вся база данных. Изменение ключа распространяется не на всю базу данных, а только на одну строку, которая достаточно легко изменяется.

С другой стороны, при таком подходе ключ всегда должен храниться в базе данных. Хранение ключей в файловой системе так, чтобы они были доступны для базы данных, может оказаться неприемлемым. Кроме того, усложняется защита файла базы данных в случае похищения как ключей, так и зашифрованных данных.

Комбинированная схема

Комбинированная схема стремится объединить высокую степень безопасности с максимальной гибкостью. Для каждой строки создается свой ключ, но в базе данных также имеется главный ключ (рис. 23.4). Процесс шифрования не сводится к простому использованию ключа, хранящегося для каждой записи. Вместо этого ключ строки объединяется с главным ключом поразрядной операцией XOR, а полученное значение используется в качестве ключа строки. Чтобы расшифровать значение, необходимо знать как ключ строки (хранящийся в базе данных), так и главный ключ (хранящийся в другом месте). Раздельное хранение этих ключей повышает уровень безопасности схемы шифрования.

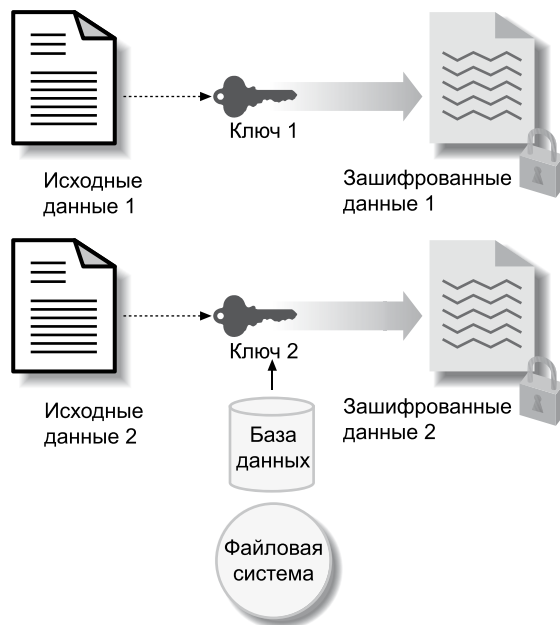


Рис. 23.3. Схема с одним ключом для каждой строки



Рис. 23.4. Комбинированная схема

Недостаток комбинированной схемы тот же, что и у стратегии с одним ключом: при потере главного ключа данные будут потеряны. Впрочем, этот риск до определенной степени компенсируется резервным копированием главного ключа.



Эта схема не эквивалентна повторному шифрованию зашифрованного значения с другим ключом. Пакет DBMS_CRYPTO не позволяет повторно шифровать зашифрованные данные, если вы попытаетесь это сделать, Oracle выдаст ошибку ORA-28233.

Рассмотрим пример использования этой схемы в реальном приложении. В код, уже приводившийся ранее при описании дешифрования, добавляется новая переменная `l_master_key` (строка 6), которая получает значение от пользователя (подстановочная переменная `&master_key`). В строках 14–18 ключ объединяется операцией XOR с главным ключом, который использовался при шифровании в строке 22, вместо переменной `l_key`.

```

/* Файл в Сети: combined_master_key.sql */
1  REM Определение переменной для хранения зашифрованного значения
2  VARIABLE enc_val varchar2(2000);
3  DECLARE
4      l_key          VARCHAR2 (2000) := '1234567890123456';
5      l_master_key   VARCHAR2 (2000) := '&master_key';
6      l_in_val       VARCHAR2 (2000) := 'Confidential Data';
7      l_mod          NUMBER
8      :=            DBMS_CRYPTO.encrypt_aes128
9                  + DBMS_CRYPTO.chain_cbc
10                 + DBMS_CRYPTO.pad_pkcs5;
11     l_enc          RAW (2000);
12     l_enc_key      RAW (2000);
13 BEGIN
14     l_enc_key :=
15         UTL_RAW.bit_xor (utl_i18n.string_to_raw (l_key, 'AL32UTF8'),
16                         utl_i18n.string_to_raw (l_master_key, 'AL32UTF8')
17                     );
18     l_enc :=
19         DBMS_CRYPTO.encrypt (utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
20                             l_mod,
21                             l_enc_key
22                         );
23     DBMS_OUTPUT.put_line ('Encrypted=' || l_enc);
24     :enc_val := RAWTOHEX (l_enc);
25 END;
26 /
27 DECLARE
28     l_key          VARCHAR2 (2000) := '1234567890123456';
29     l_master_key   VARCHAR2 (2000) := '&master_key';
30     l_in_val       RAW (2000)      := HEXTORAW (:enc_val);
31     l_mod          NUMBER
32     :=            DBMS_CRYPTO.encrypt_aes128
33                 + DBMS_CRYPTO.chain_cbc
34                 + DBMS_CRYPTO.pad_pkcs5;
35     l_dec          RAW (2000);
36     l_enc_key      RAW (2000);
37 BEGIN
38     l_enc_key :=
39         UTL_RAW.bit_xor (utl_i18n.string_to_raw (l_key, 'AL32UTF8'),
40                         utl_i18n.string_to_raw (l_master_key, 'AL32UTF8')
41                     );
42     l_dec := DBMS_CRYPTO.decrypt (l_in_val, l_mod, l_enc_key);
43     DBMS_OUTPUT.put_line ('Decrypted=' || utl_i18n.raw_to_char (l_dec));
44 END;

```

При выполнении этого блока в SQL*Plus будет получен следующий результат (обратите внимание: сначала главный ключ вводится для шифрования данных, а затем тот же главный ключ вводится при дешифровании):

```

Enter value for master_key: MasterKey0123456
old 3:   l_master_key varchar2(2000) := '&master_key';
new 3:   l_master_key varchar2(2000) := 'MasterKey0123456';
Encrypted=C2CABD4FD4952BC3ABB23BD50849D0C937D3EE6659D58A32AC69EFFD4E83F79D

```

PL/SQL procedure successfully completed.

```

Enter value for master_key: MasterKey0123456

```

```
old 3:      l_master_key varchar2(2000) := '&master_key';
new 3:      l_master_key varchar2(2000) := 'MasterKey0123456';
Decrypted=ConfidentialData
```

PL/SQL procedure successfully completed.

Программа запросила главный ключ, который был введен правильно, и программа выдала правильное значение. А что произойдет, если главный ключ задан неверно?

```
Enter value for master_key: MasterKey0123456
old 3:      l_master_key varchar2(2000) := '&master_key';
new 3:      l_master_key varchar2(2000) := 'MasterKey0123456';
Encrypted=C2CABD4FD4952BC3ABB23BD50849D0C937D3EE6659D58A32AC69EFFD4E83F79D
```

PL/SQL procedure successfully completed.

```
Enter value for master_key: MasterKey0123455
old 3:      l_master_key varchar2(2000) := '&master_key';
new 3:      l_master_key varchar2(2000) := 'MasterKey0123455';
declare
*
```

```
ERROR at line 1:
ORA-28817: PL/SQL function returned an error.
ORA-06512: at "SYS.DBMS_CRYPTO_FFI", line 67
ORA-06512: at "SYS.DBMS_CRYPTO", line 41
ORA-06512: at line 15
```

Обратите внимание на ошибку при вводе главного ключа: она привела к тому, что зашифрованные данные остались недоступными. Улучшенный механизм безопасности зависит от двух ключей, и успешное дешифрование данных возможно лишь при наличии обоих. При отсутствии главного ключа несанкционированное дешифрование становится невозможным.

Если главный ключ хранится на стороне клиента и передается по сети, потенциальный злоумышленник может воспользоваться программой-перехватчиком для похищения передаваемых данных. Существует несколько способов борьбы с перехватом данных:

- Создайте между сервером приложения и сервером базы данных виртуальную локальную сеть (VLAN), которая в значительной степени защищает передаваемый сетевой трафик.
- Главный ключ можно изменить некоторым заранее определенным способом — скажем, инвертировать все символы. В этом случае злоумышленник получит не главный ключ, а его модификацию, передаваемую по сети.
- Наконец, если вам необходимо действительно безопасное решение, защитите трафик между клиентом и сервером при помощи Oracle Advanced Security Option (поставляется за отдельную плату).

Идеальной схемы управления ключами не существует. Выбор зависит от природы приложения и попытки найти компромисс между безопасностью и простотой использования. Три основные разновидности методов управления ключами, описанные в этом разделе, станут отправной точкой для ваших собственных разработок. Возможно, у вас появится более удачная идея, которая лучше подойдет для вашей конкретной ситуации, например гибридное решение с разными ключами для важнейших таблиц.

Криптографическое хеширование

Шифрование гарантирует, что с данными смогут работать только санкционированные пользователи. Для этого критические данные преобразуются в новую форму. Однако в некоторых случаях требуется не скрыть данные, а просто защитить их от манипуляций. Классическим примером служит хранение финансовых счетов. Сами данные не


```
10      );  
11      DBMS_OUTPUT.put_line ('Hash=' || l_hash);  
12      * END;
```

Программа выводит сгенерированный хеш-код:

Hash=9222DE984C1A7DD792F680FDFD3EA05CB6CA59A9

Конечно, в реальных программах хеш-коды обычно не выводятся; они либо сохраняются в базе данных, либо передаются получателю для дальнейшей проверки.

Хеширование применяется во многих областях, не связанных с криптографией. Например, веб-приложения по своей природе не имеют состояния; сеанс приложения не обязательно соответствует «сеансу» экземпляра Oracle. Соответственно, блокировки уровня строк не гарантируют защиты от потерянных обновлений; после того как веб-страница загрузит строку, другое приложение может изменить данные. Как сеанс веб-приложения узнает об изменении загруженной ранее страницы? Одно из возможных решений заключается в генерировании и сохранении хеш-кодов данных строк. Когда в будущем приложению понадобится работать со строкой, оно выполняет повторное хеширование, сравнивает значения и быстро определяет актуальность данных.

Коды MAC

Хеширование предназначено для проверки подлинности данных, а не для их защиты от посторонних. Идея заключается в том, что вы генерируете хеш-код и сохраняете его в другом месте, отличном от места хранения самих данных. Позднее хеширование производится заново, а результат сравнивается с сохраненным значением. Однако при этом возникает небольшая проблема: что делать, если злоумышленник обновит исходные данные, выполнит хеширование и обновит сохраненный хеш-код?

Для предотвращения подобных неприятностей создается своего рода хеш-код, защищенный паролем, — он называется *кодом MAC* (Message Authentication Code). Код MAC представляет собой хеш-код, объединенный с ключом. При использовании другого ключа на основании тех же исходных данных будет сгенерирован другой код MAC. Использование ключа не позволяет злоумышленнику сгенерировать тот же код MAC, если только он не угадает ключ (поэтому не используйте очевидные значения!).

Алгоритм MAC реализуется функцией MAC пакета DBMS_CRYPTO. Функция получает три параметра:

- **src** — исходные данные (RAW).
- **key** — ключ, используемый для вычисления кода MAC.
- **typ** — используемый алгоритм. Как и при хешировании, доступны три варианта: MD4, MD5 и SHA-1. Параметр передается в виде константы пакета DBMS_CRYPTO (см. список в предыдущем разделе). Единственное отличие — замена префикса «HASH» в именах на «HMAC»; например, константа MAC для алгоритма SHA-1 называется HMAC_SH1 вместо HASH_SH1.

Следующий пример практически идентичен тому, который использовался для демонстрации хеширования, кроме добавления ключа «1234567890123456». И ключ, и исходные данные должны относиться к типу RAW; если они хранятся в переменной другого типа, их необходимо преобразовать.

```
DECLARE  
  l_in_val  VARCHAR2 (2000) := 'Critical Data';  
  l_key     VARCHAR2 (2000) := 'SecretKey';  
  l_mac     RAW (2000);  
BEGIN  
  l_mac :=
```

```

DBMS_CRYPTO.mac (src      => utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
                  typ      => DBMS_CRYPTO.hmac_sh1,
                  key      => utl_i18n.string_to_raw (l_key, 'AL32UTF8')
                );
DBMS_OUTPUT.put_line ('MAC=' || l_mac);
-- Используем другой ключ
l_key := 'Another Key';
l_mac :=
    DBMS_CRYPTO.mac (src      => utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
                    typ      => DBMS_CRYPTO.hmac_sh1,
                    key      => utl_i18n.string_to_raw (l_key, 'AL32UTF8')
                  );
DBMS_OUTPUT.put_line ('MAC=' || l_mac);
END;
```

Результат:

```

MAC=7A23524E8B665A57FE478FBE1D5BFE2406906B2E
MAC=0C0E467B588D2AD1DADE7393753E3D67FCCE800C
```

Как и предполагалось, с другим ключом для тех же исходных данных генерируется другой код MAC. Таким образом, если злоумышленник обновит код MAC, не зная ключа, он сгенерирует другой код, который не совпадет с ранее сгенерированным значением.



Приведенный пример чрезвычайно упрощен. В реальных приложениях для выполнения подобных операций используются намного более сложные ключи.

Прозрачное шифрование данных

В предыдущих разделах вы узнали, как построить инфраструктуру шифрования с нуля. Такая инфраструктура может понадобиться в том случае, если ваша организация должна выполнять требования многих правил и директив, действующих в наше время, или вы просто хотите защитить свою базу данных от потенциальных атак. Наверняка вы заметили, что в приведенных примерах построение компонентов, относящихся к шифрованию (триггеры, пакеты), было относительно простым и прямолинейным. Безусловно, самой сложной частью инфраструктуры было управление ключами шифрования. И хотя эти ключи должны быть доступны для приложений, доступ к ним необходимо ограничить для защиты их от похищения, что может быть непросто.

Начиная с Oracle Database 10g Release 2 механизм прозрачного шифрования данных (TDE, Transparent Data Encryption) сильно упрощает шифрование. Все, что для этого нужно, — пометить столбец как зашифрованный; Oracle сделает все остальное. Значение столбца обрабатывается при вводе пользователем, шифруется и сохраняется в зашифрованном формате. В будущем при запросе данных значение автоматически дешифруется, и результат возвращается пользователю, которому даже не нужно знать о выполняемом шифровании и дешифровании, — отсюда и термин «прозрачный». Все происходит в коде Oracle без необходимости применения триггеров или сложной процедурной логики.

Рассмотрим пример использования TDE: чтобы объявить столбец *SSN* таблицы *ACCOUNTS* как зашифрованный, достаточно включить в команду следующее условие:

```
ALTER TABLE accounts MODIFY (ssn ENCRYPT USING 'AES256')
```

База данных Oracle шифрует столбец *SSN* с применением алгоритма AES и 256-разрядного ключа. Ключ хранится в таблице словаря данных, но для его защиты от похищения он также шифруется с главным ключом, хранящимся в отдельном месте — *электронном бумажнике*. Бумажник по умолчанию хранится в каталоге `$ORACLE_BASE/admin/$ORACLE_SID/`

wallet; впрочем, в файле SQLNET.ORA можно выбрать другое местонахождение. Допустим, пользователь вводит данные:

```
INSERT INTO accounts (ssn) VALUES ('123456789')
```

Фактическое значение сохраняется в зашифрованном виде в файлах данных, журналах операций и их архивах, а значит, и в резервных копиях. При следующих обращениях пользователя к данным зашифрованное значение автоматически дешифруется, и пользователь получает исходное значение. Перед выполнением приведенных команд бумажник должен быть открыт администратором базы данных или администратором по безопасности.

Начиная с Oracle Database 12c в секции шифрования появился еще один параметр, обеспечивающий дополнительную защиту шифруемых данных: к каждому зашифрованному значению добавляется 20-байтовый код MAC. Если кто-то изменит зашифрованное значение, код MAC измененных данных будет отличаться от исходного, и модификация данных будет обнаружена. Однако добавление кода MAC увеличивает затраты памяти на хранение данных, что может создать проблемы в базах данных, ограниченных по занимаемому пространству. Эту функцию можно отключить:

```
ALTER TABLE accounts MODIFY (ssn ENCRYPT USING 'AES256' NOMAC)
```

Как видите, использовать TDE очень просто, поэтому возникает закономерный вопрос: насколько актуально все, что говорилось ранее о шифровании в этой главе?

КРАТКО О ШИФРОВАНИИ

- Для реализации шифрования и сопутствующих функций Oracle предоставляет два пакета: DBMS_CRYPTO (начиная с Oracle Database 10g) и DBMS_OBFUSCATION_TOOLKIT. Если вы работаете в Oracle Database 10g или более поздней версии, используйте DBMS_CRYPTO.
- Для шифрования входного значения необходимы четыре компонента: ключ, алгоритм, метод заполнения и метод сцепления.
- Чем длиннее ключ, тем труднее его подобрать, — а следовательно, тем безопаснее шифрование.
- При расшифровке данных необходимо использовать ту же комбинацию алгоритма, ключа, заполнения и сцепления, которая использовалась при шифровании.
- Основной проблемой при построении системы шифрования является управление ключами. Защита ключей в сочетании с их доступностью для приложения крайне важна для эффективной системы шифрования.
- Хешированием называется генерирование (вроде бы) случайного значения — хеш-кода — по входному значению. Входное значение не может быть восстановлено по хеш-коду. Хеш-функция для одного значения всегда выдает один и тот же хеш-код.
- Код MAC по сути не отличается от хеш-кода, если не считать того, что в процессе генерирования используется ключ.

Вовсе нет! Область применения TDE ограничена защитой файлов базы данных от возможного похищения посредством шифрования конфиденциальной информации с минимальными усилиями. Однако следует обратить особое внимание на слово «прозрачный» в названии технологии — то есть автоматически выполняется как шифрование, так и дешифрование данных. Oracle не различает пользователей в контексте базы данных. Когда пользователь обращается с запросом к данным, Oracle предоставляет текстовое значение независимо от того, кто выдал запрос.

Во многих случаях требуется строить более сложные системы, в которых доступ к текстовому значению открывается только в том случае, если пользователь, выдавший запрос, обладает привилегиями для его просмотра; в остальных случаях должно возвращаться зашифрованное значение. Выполнить это требование средствами TDE не удастся, потому что TDE расшифровывает все данные без разбора. Впрочем, желаемой цели можно добиться построением собственной инфраструктуры с применением методов, описанных в этой главе.

У TDE имеются свои ограничения. Во-первых, технология TDE не может использоваться для шифрования столбца внешнего ключа; во многих бизнес-приложениях это может стать серьезным препятствием. Кроме того, для столбцов, защищенных TDE, могут создаваться только индексы B-деревьев. При самостоятельной реализации шифрования средствами PL/SQL такие ограничения отсутствуют.

Принимая решение о том, подойдет ли TDE для ваших целей, также необходимо принять во внимание другой аспект — автоматизацию. В TDE электронный бумажник (в котором хранится главный ключ) должен быть открыт администратором базы данных, для чего используется команда следующего вида:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN AUTHENTICATED BY "pooh";
```

Здесь "pooh" — пароль к бумажнику. Если файлы базы данных (или журналы операций, или резервные копии этих файлов) будут похищены, то зашифрованные столбцы останутся недоступными — вор не знает пароль, который позволит ему открыть бумажник.

После каждого запуска базы данных вставка или обращение к зашифрованным столбцам станут возможны лишь после того, как бумажник будет явно открыт администратором. Если бумажник закрыт, операции вставки и попытки обращения к этим столбцам завершатся неудачей. После открытия базы данных необходимо выполнить одно лишнее действие; кроме того, человек, открывающий базу данных, должен знать пароль к бумажнику.

В принципе для упрощения и автоматизации процесса можно создать триггер запуска базы данных, который вызывает команду ALTER SYSTEM (см. выше). Но в этом случае вы лишаете единственной защиты свой бумажник — а следовательно, и зашифрованные столбцы. Итак, при использовании TDE такие триггеры создавать никогда не следует, а вы должны быть готовы к выполнению лишней операции при каждом запуске базы данных. Однако самостоятельно реализованная инфраструктура шифрования станет доступна одновременно с базой данных; никакие дополнительные действия не потребуются, а вам не придется запоминать и вводить пароли от бумажника.

Короче говоря, область применения TDE ограничена. Эта технология предоставляет простые и быстрые средства для шифрования файлов данных, журналов операций и резервных копий. Тем не менее она не защищает данные посредством ограничения доступа в зависимости от пользователя; данные всегда дешифруются при обращении. Если вы хотите более точно управлять процессом дешифрования, то вам придется строить собственную инфраструктуру.

Прозрачное шифрование табличного пространства

Итак, основные недостатки TDE (и в меньшей степени — пользовательских реализаций шифрования) в отношении производительности приложения:

- TDE не позволяет использовать индексы для диапазонных запросов, поскольку табличные данные не коррелируют со значениями индекса. Пользовательская реализация шифрования предоставляет ограниченные возможности для использования индексов.

- Запросы к шифрованным данным требуют дешифрования, что приводит к существенным дополнительным затратам вычислительных ресурсов.

По этим причинам при разработке реальных приложений технология TDE часто отвергается как неприемлемая, а обширные требования к пользовательским реализациям шифрования на базе DBMS_CRYPTO создают изрядные трудности во многих организациях. Для решения этих проблем в Oracle Database 11g появилась новая технология *прозрачного шифрования табличного пространства* (TTE, Transparent Tablespace Encryption). Она позволяет включить шифрование для всего табличного пространства, а не для отдельных таблиц. Пример создания шифрованного табличного пространства:

```
TABLESPACE securets1
  DATAFILE '+DG1/securets1_01.dbf'
  SIZE 10M
  ENCRYPTION USING 'AES128'
  DEFAULT STORAGE (ENCRYPT)
```

Все объекты, создаваемые в этом табличном пространстве, должны преобразовываться в зашифрованный формат по алгоритму AES с использованием 128-разрядного ключа. Для этого необходимо заранее создать бумажник и открыть его так, как описано в предыдущем разделе. Ключ шифрования хранится в таблице ENC\$ в зашифрованном виде, а ключ к этому шифрованию хранится в бумажнике (как и в случае TDE). Конечно, бумажник должен быть открыт до создания табличного пространства.

Как шифрование табличного пространства поможет решить проблемы шифрования уровня таблиц, спросите вы? Принципиальное отличие между двумя технологиями заключается в том, что данные в табличном пространстве шифруются только на диске; сразу же после чтения данные дешифруются и помещаются в буферный кэш SGA в виде простого текста. Операции сканирования индекса выполняются с буферным кэшем, тем самым решается проблема несоответствий зашифрованных данных. Кроме того, поскольку данные дешифруются и помещаются в буферный кэш только один раз (по крайней мере до их устаревания), дешифрование происходит только один раз, а не при каждом обращении к данным. Соответственно, в то время, пока данные остаются в SGA, шифрование не снижает производительность. Достигаются сразу обе цели — безопасность посредством шифрования и минимальное влияние на производительность.

Итак, проблема решена, и с TTE отпадает надобность в пользовательских процедурах шифрования, приводившихся ранее? Вовсе нет!

При шифровании табличного пространства шифруются *все* объекты — индексы и таблицы, независимо от того, нужно их шифровать или нет, если вам потребовалось шифровать все данные в табличном пространстве или хотя бы их большинство. А если шифроваться должна лишь малая часть общего объема данных? С применением TTE на производительность вашего приложения будет влиять существенно больший объем данных, чем это действительно необходимо. База данных Oracle сокращает последствия шифрования, но не может полностью избежать их. В результате вам, может, все равно придется самостоятельно реализовать избирательное шифрование данных в таблицах своего приложения.

Кроме того, зашифрованные табличные пространства могут только создаваться; вы не сможете преобразовать существующее табличное пространство в зашифрованную форму (или шифрованное табличное пространство в обычное). Вместо этого придется создать зашифрованное табличное пространство и переместить в него объекты. При внедрении шифрования в существующую базу данных решение на базе TTE может оказаться неприемлемым из-за гигантских объемов многих рабочих баз данных. Пользовательское шифрование позволяет точно управлять тем, какие данные будут шифроваться (и дешифроваться) вашим приложением.

Безусловно, пользовательская реализация шифрования находит свое место в реальных приложениях. Прозрачное шифрование табличного пространства реализуется намного быстрее и проще, но вы должны убедиться в том, что «тотальное» шифрование, основанное на методе «грубой силы», подходит для вашего приложения.

ШИФРОВАНИЕ И EXADATA

Exadata, мощная машина для работы с базами данных от компании Oracle, добилась известности в области аппаратных решений Oracle Database. По сути это обычная версия Oracle Database, работающая на специализированном оборудовании, поэтому все концепции, описанные в этой главе, применимы к Exadata без каких-либо уточнений. Тем не менее одна возможность, связанная с шифрованием, оказывается настолько мощной, что о ней стоит упомянуть особо. В отличие от обычных механизмов хранения данных, система хранения данных Exadata ведет себя «интеллектуально» в двух отношениях. Во-первых, она может располагать информацией о распределении данных по дискам, что позволяет ей фильтровать неактуальные блоки прямо на дисках. Во-вторых, при содействии программного обеспечения — Exadata Storage Server — она может выполнять некоторые операции (применение функции MIN или MAX) без участия базы данных. Перемещение некоторых трудоемких операций, обычно выполняемых на уровне базы данных, на уровень хранения информации приводит к общему снижению объема ввода/вывода и передачи данных между ячейками и узлами базы данных; соответственно Exadata демонстрирует серьезные результаты в отношении производительности. Конечно, не все функции можно передать на уровень хранения, но одной из таких задач может быть шифрование. База данных выполняет сжатие, но дешифрование выполняется ячейками системы хранения.

Безопасность уровня строк

Введенный в Oracle8i механизм безопасности уровня строк, или *RLS* (Row Level Security), позволяет определять для таблиц (и конкретных типов операций с таблицами) политики безопасности, ограничивающие возможности просмотра и изменения строк таблиц пользователями. Большая часть соответствующей функциональности реализуется встроенным пакетом `DBMS_RLS`.

В Oracle в течение многих лет безопасность обеспечивалась на уровне таблиц, и в определенной степени на уровне столбцов. Пользователям могли предоставляться привилегии, разрешающие или ограничивающие доступ отдельными столбцами. Триггеры `INSTEAD OF` (см. главу 19) позволяли ограничить обновление таблиц из представлений. Все эти привилегии основывались на одном предположении: безопасность может быть реализована простым ограничением доступа к некоторым таблицам и столбцам. Но когда пользователь получает доступ к таблице, он видит все ее строки. А если потребуется ограничить видимость строк таблицы на основании таких критериев, как уровень полномочий текущего пользователя, или других характеристик конкретного приложения?

Для примера возьмем демонстрационную таблицу `EMP` из схемы `HR`, входящей в поставку Oracle. Таблица содержит 14 строк данных с первичными ключами (кодами работников) от 7369 до 7934.

Допустим, вы предоставили пользователю `Loga` доступ к таблице, но хотите ограничить его так, чтобы пользователь мог видеть и изменять строки только тех пользователей, у которых поле `COMM` отлично от `NULL`.

Проблему можно решить созданием представления на базе таблицы, но что если пользователь должен иметь доступ к самой таблице? В некоторых случаях доступ к таблице может быть оправданным — например, если пользователю потребуется создавать хранящиеся программные блоки для работы с таблицей. Реализация с представлением для этого просто не подойдет.

В таких случаях на помощь приходит RLS. Фактически вы приказываете Oracle ограничить набор строк, видимых пользователю, на основании определяемого вами правила. Пользователь никак не сможет обойти это ограничение.



В документации Oracle RLS также иногда обозначается сокращениями VPD (Virtual Private Database) и FGAC (Fine-Grained Access Control).

Например, если активизировать RLS для таблицы EMP с описанным выше правилом, то при вводе запроса

```
SELECT * FROM emp
```

пользователь Lora увидит только четыре строки (а не 14!), хотя сам запрос не включает секцию WHERE:

| | | | | | | | |
|------|--------|----------|------|-----------|-------|-------|----|
| 7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 | 1,600 | 300 | 30 |
| 7521 | WARD | SALESMAN | 7698 | 22-FEB-81 | 1,250 | 500 | 30 |
| 7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 | 1,250 | 1,400 | 30 |
| 7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 | 1,500 | 0 | 30 |

Аналогичным образом, при обновлении таблицы без условия WHERE будут обновлены только видимые пользователю записи:

```
SQL> UPDATE hr.emp SET comm = 100
      2 /
```

4 rows updated.

Остальные 10 записей словно не существуют для пользователя Lora. Реализация RLS основана на включении *предиката* (условия WHERE) в любые DML-инструкции, выполняемые пользователем с таблицей. В нашем примере запрос SELECT * FROM EMP автоматически преобразуется к виду

```
SELECT * FROM emp WHERE comm IS NOT NULL
```

Чтобы для таблицы действовали автоматические ограничения доступа, необходимо определить для нее политику RLS. Политика определяет наличие возможных ограничений при обращении к данным. Допустим, вы хотите, чтобы операции UPDATE были недоступны для пользователей, а операции SELECT выполнялись без каких-либо ограничений или чтобы доступ к SELECT ограничивался только при выборке определенного столбца (скажем, SALARY). Все эти инструкции включаются в политику. Политика связывается с функцией, которая генерирует предикат (COMM IS NOT NULL в нашем примере), применяемый к запросам.

Итак, на верхнем уровне абстракции RLS состоит из трех основных компонентов:

- **Политика** — декларативная команда, которая определяет, когда и как применяются ограничения: при выборке, вставке, удалении, обновлении или в комбинациях этих операций.
- **Функция политики** — функция PL/SQL, вызываемая при выполнении условий, заданных в политике.

- **Предикат** — строка, которая генерируется функцией политики, а затем включается в SQL-инструкции пользователей для определения ограничивающих условий.

На концептуальном уровне схема действия RLS показана на рис. 23.5. Политика представляет собой «фильтр» для отбора строк таблицы. Если строка удовлетворяет предикату, она проходит через фильтр; в противном случае строка остается невидимой для пользователя.

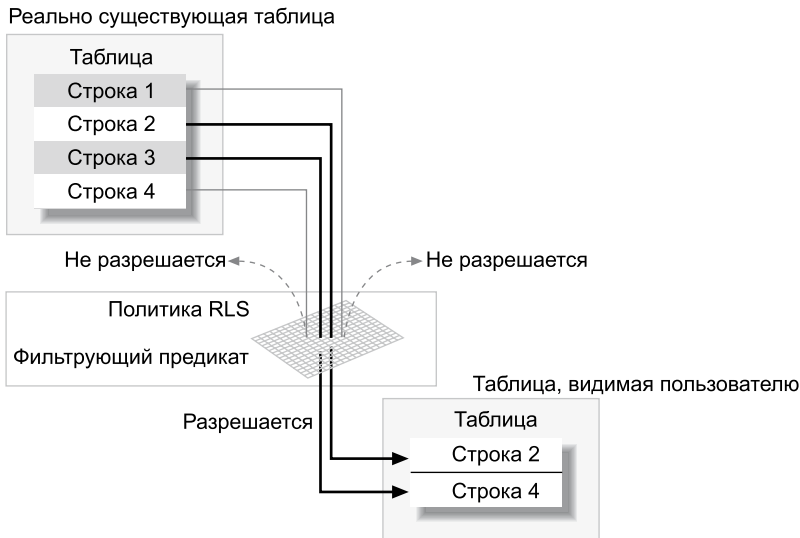


Рис. 23.5. Инфраструктура безопасности уровня строк

Зачем изучать RLS?

На первый взгляд тема безопасности уровня строк представляет интерес для администраторов баз данных и администраторов безопасности, а не для разработчиков PL/SQL и даже проектировщиков архитектуры приложений. Зачем разработчику PL/SQL изучать ее?

- **В наши дни безопасность — дело общее.** Механизм RLS предназначен для обеспечения безопасности, которая традиционно относилась к ведению администраторов баз данных. Однако в XXI веке безопасность становится все более насущной проблемой. Бесчисленные законы, правила и рекомендации ограничивают наши приложения, и разработчикам все чаще приходится учитывать аспекты безопасности при построении программ. В частности, проектировщик архитектуры приложения должен хорошо разбираться в применении RLS на ранних стадиях проектирования.
- **RLS — не только безопасность.** По сути, концепция RLS определяет некий «фильтр», находящийся под управлением разработчика функции. Допустим, вам приходится обеспечивать поддержку стороннего приложения с двумя функциональными областями, данные которых хранятся в одной таблице; вам потребовалось внести изменения в запросы приложения, чтобы эти функциональные области более четко отделялись друг от друга. Однако для этого придется вносить изменения в код приложения, что может быть нежелательно. На помощь приходит RLS. Этот механизм позволяет реализовать логическое разбиение строк таблицы, чтобы два приложения «видели» разные наборы данных. Безусловно, такой подход упрощает задачу разработчика и сопровождение системы в целом.

- **RLS позволяет делать то, что не может быть сделано другими средствами.** Не забывайте, что RLS применяет к запросам предикат, генерируемый функцией. Если вы сгенерируете предикат `1=2`, результат которого всегда равен `FALSE`, что выдаст ваш запрос? Ничего, потому что условие `WHERE` всегда ложно. Таким образом, определение политики `1=2` для команд `DELETE`, `INSERT` и `UPDATE`, но не для `SELECT`, фактически запрещает пользователю изменять таблицу, но не мешает чтению данных. Oracle по умолчанию позволяет устанавливать ограничение доступа «только для чтения» для табличных пространств, но не для конкретных таблиц. RLS позволяет сделать доступной только для чтения отдельную таблицу.

Другие средства не подходят: если просто лишить пользователя привилегий `UPDATE` или `INSERT`, процедуры станут недействительными. Если определить процедуру с использованием модели прав создающего (см. главу 24), то вам не удастся отозвать привилегии у конкретного пользователя.

Несколько простых примеров помогут вам понять принципы работы RLS.

Простой пример использования RLS

В этом примере используется таблица `EMP`, уже упоминавшаяся ранее в этой главе. В результате применения RLS пользователи не должны видеть строки с информацией о работниках, оклад которых превышает \$1500. Чтобы реализовать политику, мы должны определить следующие компоненты:

- Предикат, который будет автоматически включаться в команды SQL пользователей.
- Функция политики, генерирующая этот предикат.
- Политика, которая вызывает функцию и применяет предикат (прозрачно с точки зрения пользователя).

В нашем примере предикат выглядит так:

```
SALARY <= 1500
```

Функции политики:

```
FUNCTION authorized_emps (
  p_schema_name IN VARCHAR2,
  p_object_name IN VARCHAR2
)
  RETURN VARCHAR2
IS
  l_return_val VARCHAR2 (2000);
BEGIN
  l_return_val := 'SAL <= 1500';
  RETURN l_return_val;
END authorized_emps;
```

При выполнении эта функция возвращает строку `SAL <= 1500`. Следующий фрагмент кода это наглядно доказывает:

```
DECLARE
  l_return_string VARCHAR2 (2000);
BEGIN
  l_return_string := authorized_emps ('X', 'X');
  DBMS_OUTPUT.put_line ('Return String = "' || l_return_string || '"');
END;
```

Результат:

```
Return String = "SAL <= 1500"
```

Резонно спросить, зачем передавать аргументы, если функция всегда возвращает одно и то же значение? Смысл этого требования RLS будет объяснен позднее.

Наконец, мы создаем политику при помощи функции `ADD_POLICY` из встроенного пакета `DBMS_RLS`:

```

1 BEGIN
2     DBMS_RLS.add_policy (object_schema => 'HR',
3                          object_name   => 'EMP',
4                          policy_name   => 'EMP_POLICY',
5                          function_schema => 'HR',
6                          policy_function => 'AUTHORIZED_EMPS',
7                          statement_types => 'INSERT, UPDATE, DELETE, SELECT'
8                          );
9 END;
```

Политика с именем `EMP_POLICY` (строка 4) добавляется для таблицы `EMP` (строка 4), принадлежащей схеме `HR` (строка 2). Политика применяет фильтр, генерируемый функцией `AUTHORIZED_EMPS` (строка 6), принадлежащей схеме `HR` (строка 5), при каждом выполнении пользователем операции `INSERT`, `UPDATE`, `DELETE` или `SELECT` (строка 7). Функция `AUTHORIZED_EMPS`, которая создает и возвращает предикатные строки, применяемые к запросам, была написана ранее.

После установления этой политики при выборке данных из таблицы или их обновлении пользователь сможет работать только с теми записями, для которых выполняется условие `SAL <= 1500`.

В зависимости от потребностей приложения функция политики может определяться произвольно, но она должна удовлетворять некоторым правилам:

- Функция политики может быть отдельной или пакетной функцией, но не может быть процедурой.
- Функция политики должна возвращать значение типа `VARCHAR2`, которое будет применяться как предикат. В частности, из этого следует, что длина предиката не может превышать 32 767 байт.
- Функция политики должна получать ровно два входных параметра, которые передаются в следующем порядке:
 - *schema* — схема-владелец таблицы, для которой определяется политика;
 - *object_name* — имя объекта, идентифицирующее таблицу или представление.



Чтобы снять все ограничения доступа, укажите функцию политики, которая возвращает в качестве предиката один из следующих вариантов:

- `NULL`.
- `1=1` или другое выражение, результат которого всегда равен `TRUE`.

Так как возвращаемое значение должно относиться к типу `VARCHAR2`, просто вернуть `TRUE` не удастся.

Аналогичным образом устанавливается ограничение для всех строк: выбирается предикат, результат которого всегда равен `FALSE` — например, `1=2`.

Для таблицы можно определить несколько политик RLS. Приоритетов, то есть фиксированного порядка применения политик к запросам, не существует. К SQL-запросу присоединяются предикаты, возвращаемые всеми политиками.

Информацию обо всех политиках, определенных для таблицы, можно получить из представления словаря данных `DBA_POLICIES`: имя политики, объект, для которого она определена (и его владелец), имя функции политики (и ее владелец) и многие другие сведения.

Начиная с Oracle10g, параметр `statement_types` может принимать дополнительное значение `INDEX`. В этом случае доступ к строкам ограничивается даже при создании индексов.

Допустим, вы хотите создать индекс по ключу-функции для столбца SAL; сценарию создания индекса для этого понадобятся все значения столбца, то есть фактически система безопасности будет обойдена. Пример включения значения INDEX в параметр:

```
1 BEGIN
2     DBMS_RLS.add_policy (object_schema => 'HR',
3                          object_name    => 'EMP',
4                          policy_name    => 'EMP_POLICY',
5                          function_schema => 'HR',
6                          policy_function => 'AUTHORIZED_EMPS',
7                          statement_types => 'INSERT, UPDATE, DELETE, SELECT, INDEX'
8                          );
9 END;
```

При попытке создания индекса по ключу-функции будет выдана ошибка:

ORA-28133: full table access is restricted by fine-grained security.

Итак, вы научились создавать политики. Для удаления политик используется функция DROP_POLICY из пакета DBMS_RLS. Например, удаление политики EMP_POLICY осуществляется следующей командой:

```
BEGIN
    DBMS_RLS.drop_policy (object_schema => 'HR',
                          object_name    => 'EMP',
                          policy_name    => 'EMP_POLICY'
                          );
END;
```

Обратите внимание: политики не являются объектами схемы базы данных, то есть у них нет объекта-владельца. Любой пользователь с привилегией EXECUTE для пакета DBMS_RLS может создать политику. Аналогичным образом, любой пользователь с привилегией EXECUTE может удалить политику. Следовательно, привилегии EXECUTE для этого пакета должны предоставляться только тем пользователям, которым они действительно необходимы.

Рассмотрим один нюанс: вместо других столбцов пользователь обновляет столбец SAL, который используется в предикате. Интересно увидеть результат:

```
SQL> UPDATE hr.emp SET sal = 1200;
```

7 rows updated.

```
SQL> UPDATE hr.emp SET sal = 1100;
```

7 rows updated.

Как и ожидалось, обновляются только семь строк. Теперь давайте изменим обновляемое значение:

```
SQL> UPDATE hr.emp SET sal = 1600;
```

7 rows updated.

```
SQL> UPDATE hr.emp SET sal = 1100;
```

0 rows updated.

Естественно, при втором обновлении строки остаются неизменными, потому что первое обновление сделало все строки в таблице недоступными для пользователя — ведь политика RLS устанавливает фильтрующий предикат SAL <= 1500.

Возникает весьма запутанная ситуация: сами обновления данных могут изменять видимость строк таблицы. В ходе разработки это может породить ошибки или по крайней мере

создать определенную непредсказуемость в работе программы. Чтобы предотвратить путаницу, давайте воспользуемся другим параметром `DBMS_RLS.ADD_POLICY`, который называется `update_check`. Следующий пример показывает, к каким последствиям приводит присваивание этому параметру `TRUE` при создании политики для таблицы:

```
BEGIN
  DBMS_RLS.add_policy (object_name      => 'EMP',
                      policy_name       => 'EMP_POLICY',
                      function_schema   => 'HR',
                      policy_function    => 'AUTHORIZED_EMPS',
                      statement_types    => 'INSERT, UPDATE, DELETE, SELECT',
                      update_check      => TRUE
                      );
END;
```

После того как данная политика будет установлена для таблицы, при выполнении того же обновления пользователь `Lora` получит сообщение об ошибке:

```
SQL> UPDATE hr.emp SET sal = 1600;
UPDATE hr.emp SET sal = 1600
*
ERROR at line 1:
ORA-28115: policy with check option violation
```

Ошибка `ORA-28115` возникает из-за того, что новая политика запрещает обновление столбцов значениями, изменяющими видимость строк под действием RLS. Но допустим, столбец `SAL` обновляется значением, не изменяющим видимости строк:

```
SQL> UPDATE hr.emp SET sal = 1200;

7 rows updated.
```

Так как с новым значением столбца `SAL` — 1200 — все семь строк по-прежнему остаются видимыми, такое обновление разрешается.



Присвойте параметру `update_check` значение `TRUE` при определении политики, чтобы избежать непредсказуемого (по крайней мере на первый взгляд) поведения приложения.

Статические и динамические политики

В предыдущем примере использовалась политика, которая возвращала предикатную строку с константой (`SAL <= 1500`). В реальных приложениях такие ситуации встречаются редко (разве что в специализированных приложениях — например, в системах складского учета). Чаще фильтры приходится определять в зависимости от пользователя, выдающего запрос. Например, приложение для отдела кадров может ограничивать пользователя просмотром записей, относящихся к его отделу. Такое требование является *динамическим*, так как условие проверки вычисляется заново для каждого пользователя, работающего с приложением.

И это не единственное правило, которое необходимо применить в данной ситуации. Таблица защищена политикой RLS, запрещающей пользователям просмотр всех записей. Но что если выборку выполнит сам владелец таблицы (пользователь `HR`)? Он тоже увидит только ограниченное подмножество записей, а это неправильно: владелец должен видеть все записи без исключения. Возможны два варианта:

- Предоставить специальные привилегии пользователю `HR`, чтобы политика RLS на него не распространялась.

- Определить функцию политики таким образом, что если вызывающий пользователь является владельцем схемы, ограничительный предикат для него не действовал.

В первом случае функцию политики изменять не нужно — администратор базы данных может предоставить пользователю HR необходимые привилегии:

```
GRANT EXEMPT ACCESS POLICY TO hr;
```

Эта инструкция освобождает пользователя HR от действия каких-либо политик RLS. Поскольку политики перестают действовать независимо от того, для какой таблицы они определены, этот способ следует применять крайне осторожно (а лучше вообще его избегать).

Во втором способе проблема решается посредством изменения функции политики. Вот как может выглядеть соответствующая функция:

```
1  FUNCTION authorized_emps (
2      p_schema_name  IN   VARCHAR2,
3      p_object_name  IN   VARCHAR2
4  )
5      RETURN VARCHAR2
6  IS
7      l_deptno        NUMBER;
8      l_return_val    VARCHAR2 (2000);
9  BEGIN
10     IF (p_schema_name = USER)
11     THEN
12         l_return_val := NULL;
13     ELSE
14         SELECT deptno
15             INTO l_deptno
16             FROM emp
17             WHERE ename = USER;
18
19         l_return_val := 'DEPTNO = ' || l_deptno;
20     END IF;
21
22     RETURN l_return_val;
23 END;
```

Рассмотрим этот код более подробно.

| Строки | Описание |
|--------|--|
| 10 | Проверяем, является ли вызывающий пользователь владельцем таблицы. Если является — вместо предиката возвращается NULL, то есть ограничения доступа к таблице отсутствуют |
| 14–19 | Определение номера отдела пользователя и конструирование предиката в форме «DEPTNO = номер_отдела» |
| 22 | Функция возвращает предикат вызывающей стороне |

У этого метода есть одно интересное побочное преимущество. Функция политики возвращает ограничивающий предикат с ограничением по DEPTNO, поэтому данная политика применима к любой таблице со столбцом DEPTNO.

Приведенный пример демонстрирует крайний случай динамической политики. При возвращении каждой строки политика выполняет функцию, проверяет предикат и решает, стоит или нет пропускать строку. Несомненно, такой подход сопряжен со значительными затратами, потому что базе данных приходится каждый раз проходить полный цикл «разбор — выполнение — выборка».

Если предикат остается неизменным, производительность приложения можно повысить за счет ликвидации избыточных вызовов функции. Начиная с Oracle9i процедура ADD_POLICY поддерживает параметр static_policy, по умолчанию равный FALSE. Если параметр равен TRUE, то функция политики выполняется только один раз в начале


```

22             policy_type          => DBMS_RLS.SHARED_STATIC
23         );
24     END;

```

Объявляя одну политику для обеих таблиц, мы приказываем базе данных кэшировать результат функции политики, а затем использовать его при последующих вызовах.

Контекстная политика

Как говорилось ранее, статические политики при всей своей эффективности сопряжены с определенным риском; так как функция не выполняется заново при каждой проверке, результаты могут оказаться неожиданными. По этой причине Oracle предоставляет другую разновидность политик — контекстные политики, которые выполняют функцию политики только при изменении контекста приложения в сеансе (см. далее раздел «Контексты приложений»). Пример использования таких политик:

```

1  BEGIN
2      DBMS_RLS.drop_policy (object_schema => 'HR',
3                          object_name    => 'DEPT',
4                          policy_name    => 'EMP_DEPT_POLICY'
5                      );
6      DBMS_RLS.add_policy (object_schema => 'HR',
7                          object_name    => 'DEPT',
8                          policy_name    => 'EMP_DEPT_POLICY',
9                          function_schema => 'RLSOWNER',
10                         policy_function => 'AUTHORIZED_EMPS',
11                         statement_types => 'SELECT, INSERT, UPDATE,
DELETE',
12                         update_check   => TRUE,
13                         policy_type    => DBMS_RLS.CONTEXT_SENSITIVE
14                     );
15     DBMS_RLS.add_policy (object_schema => 'HR',
16                         object_name    => 'EMP',
17                         policy_name    => 'EMP_DEPT_POLICY',
18                         function_schema => 'RLSOWNER',
19                         policy_function => 'AUTHORIZED_EMPS',
20                         statement_types => 'SELECT, INSERT, UPDATE,
DELETE',
21                         update_check   => TRUE,
22                         policy_type    => DBMS_RLS.CONTEXT_SENSITIVE
23                     );
24     END;

```

Контекстная политика (DBMS_RLS.CONTEXT_SENSITIVE) обычно уступает по производительности типу SHARED_STATIC, но превосходит DYNAMIC. Давайте проанализируем затраты времени для конкретного запроса. Для хронометража будет использоваться встроенный таймер DBMS_UTILITY.GET_CPU_TIME.

```

DECLARE
    l_start_time  PLS_INTEGER;
    l_count       PLS_INTEGER;
BEGIN
    l_start_time := DBMS_UTILITY.get_time;
    SELECT COUNT ( * )
    INTO l_count
    FROM hr.emp;
    DBMS_OUTPUT.put_line (DBMS_UTILITY.get_time - l_start_time);
END;

```

Разность между возвращаемыми значениями функции для конечного и начального момента составляет время в сотых долях секунды. В следующей таблице приведены значения времени отклика для разных типов политик.

| Тип политики | Время отклика (1/100 секунды) |
|--------------|-------------------------------|
| Динамическая | 133 |
| Контекстная | 84 |
| Статическая | 37 |

Общая контекстная политика

Общие контекстные политики сходны с контекстными политиками, но с ними одна политика используется для нескольких объектов, как было показано ранее для общих статических политик.

СТРАТЕГИИ ПЕРЕХОДА НА НОВЫЕ ТИПЫ ПОЛИТИК ORACLE10G/11G

При переходе с Oracle9i на Oracle10g и выше рекомендуется действовать так:

1. Сначала использовать тип политики по умолчанию (динамическая).
2. После завершения перехода попробуйте воссоздать политику в контекстном варианте. Тщательно протестируйте результаты со всеми возможными сценариями, чтобы устранить все потенциальные проблемы с кэшированием.
3. Для тех политик, которые могут быть преобразованы в статические, выполните преобразование и проведите тщательное тестирование.

Использование столбцовой модели RLS

Вернемся к примеру приложения отдела кадров из предыдущих разделов. При проектировании политики я руководствовался требованием, согласно которому каждый пользователь может видеть только данные работников своего отдела. Тем не менее в некоторых ситуациях ограничения этой политики могут оказаться слишком жесткими.

Предположим, вы хотите защитить данные, чтобы посторонние не интересовались информацией о чужих окладах. Возьмем следующие два запроса:

```
SELECT empno, sal FROM emp
SELECT empno FROM emp
```

Первый запрос выводит оклады работников — ту самую информацию, которую вы пытаетесь защитить. В данном случае должны выводиться только данные работников из отдела пользователя. Второй запрос выводит только коды работников. Нужно ли отфильтровать и их, чтобы запрос выводил только коды работников того же отдела?

Ответ зависит от политики безопасности, действующей в вашей организации. Возможно, имеются веские причины для вывода вторым запросом всех работников независимо от того, какому отделу они принадлежат. Будет ли механизм RLS эффективным в таких ситуациях?

В Oracle9i Database RLS не поможет; однако в Oracle Database 10g и выше появился параметр `ADD_POLICY` с именем `sec_relevant_cols`, который упрощает решение этой задачи. Например, в описанном сценарии фильтр должен применяться только при выборе столбцов `SAL` и `COMM`, но не других столбцов. Политику можно записать следующим образом (обратите внимание на новый параметр):

```
BEGIN
/* Начинаем с удаления политики. */
DBMS_RLS.drop_policy (object_schema    => 'HR',
```

```

        object_name      => 'EMP',
        policy_name      => 'EMP_POLICY'
    );
/* Добавление политики. */
DBMS_RLS.add_policy (object_schema      => 'HR',
                    object_name        => 'EMP',
                    policy_name        => 'EMP_POLICY',
                    function_schema    => 'RLSOWNER',
                    policy_function     => 'AUTHORIZED_EMPS',
                    statement_types    => 'INSERT, UPDATE, DELETE, SELECT',
                    update_check       => TRUE,
                    sec_relevant_cols  => 'SAL, COMM'
                );
END;
```

После назначения этой политики запросы к HR.EMP будут выдавать разные результаты:

SQL> -- безвредный запрос, в выборку включается только EMPNO

SQL> SELECT empno FROM hr.emp; ... строки ...

14 rows selected.

SQL> -- конфиденциальный запрос, в выборку включается SAL

SQL> SELECT empno, sal FROM hr.emp; ... строки ...

6 rows selected.

При выборке столбца SAL вступает в действие политика RLS, предотвращающая вывод всех строк; она отфильтровывает строки, в которых значение DEPTNO отлично от 30 (значение DEPTNO пользователя, выполняющего запрос).

Столбцовая модель распространяется не только на список выборки, но и на любые обращения к столбцам, прямые или косвенные. Возьмем следующий запрос:

```

SQL> SELECT deptno, count(*)
2   FROM hr.emp
3  WHERE sal > 0
4  GROUP BY deptno;
```

| DEPTNO | COUNT(*) |
|--------|----------|
| 30 | 6 |

Здесь столбец SAL упоминается в секции WHERE, поэтому политика RLS оставляет только записи, относящиеся к отделу 30. Другой пример:

```

SQL> SELECT *
2   FROM hr.emp
3  WHERE deptno = 10;
```

no rows selected

Здесь вместо прямого обращения к столбцу SAL используется косвенное обращение через секцию SELECT *, поэтому политика RLS отфильтровывает записи всех отделов, кроме 30. Так как запрос вызывается для отдела 10, он не возвращает ни одну строку.

Теперь рассмотрим несколько иную ситуацию: в предыдущем случае я защитил значения столбцов SAL от отображения в строках, просмотр которых не разрешен пользователю. Однако при этом был отключен вывод всей строки, не только этого конкретного столбца. Теперь допустим, что в соответствии с новыми требованиями маскироваться должен только столбец, а не вся строка, а все остальные столбцы, не содержащие конфиденциальной информации, должны отображаться. Возможно ли это?

Да, и притом легко — в этом вам поможет другой параметр ADD_POLICY, который называется sec_relevant_cols_opt. Воссоздадим политику, задав этому параметру значение DBMS_RLS.ALL_ROWS:

```
BEGIN
  DBMS_RLS.drop_policy (object_schema => 'HR',
                        object_name   => 'EMP',
                        policy_name   => 'EMP_POLICY'
                       );
  DBMS_RLS.add_policy (object_schema => 'HR',
                        object_name   => 'EMP',
                        policy_name   => 'EMP_POLICY',
                        function_schema => 'RLSOWNER',
                        policy_function => 'AUTHORIZED_EMPS',
                        statement_types => 'SELECT',
                        update_check   => TRUE,
                        sec_relevant_cols => 'SAL, COMM',
                        sec_relevant_cols_opt => DBMS_RLS.all_rows
                       );
END;
```

Если выдать тот же запрос сейчас, результаты будут другими:

```
SQL> -- Вместо NULL в выходных данных отображается "?" in the output.
SQL> SET NULL ?
SQL> SELECT *
  2 FROM hr.emp
  3 ORDER BY deptno
  4 /
```

| EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO |
|-------|--------|-----------|------|-----------|-------|-------|--------|
| 7782 | CLARK | MANAGER | 7839 | 09-JUN-81 | ? | ? | 10 |
| 7839 | KING | PRESIDENT | ? | 17-NOV-81 | ? | ? | 10 |
| 7934 | MILLER | CLERK | 7782 | 23-JAN-82 | ? | ? | 10 |
| 7369 | SMITH | CLERK | 7902 | 17-DEC-80 | ? | ? | 20 |
| 7876 | ADAMS | CLERK | 7788 | 12-JAN-83 | ? | ? | 20 |
| 7902 | FORD | ANALYST | 7566 | 03-DEC-81 | ? | ? | 20 |
| 7788 | SCOTT | ANALYST | 7566 | 09-DEC-82 | ? | ? | 20 |
| 7566 | JONES | MANAGER | 7839 | 02-APR-81 | ? | ? | 20 |
| 7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 | 1,600 | 300 | 30 |
| 7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 | 2,850 | ? | 30 |
| 7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 | 1,250 | 1,400 | 30 |
| 7900 | JAMES | CLERK | 7698 | 03-DEC-81 | 950 | ? | 30 |
| 7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 | 1,500 | 0 | 30 |
| 7521 | WARD | SALESMAN | 7698 | 22-FEB-81 | 1,250 | 500 | 30 |

14 rows selected.



Будьте внимательны при использовании этой возможности, потому что в некоторых случаях она приводит к неожиданным результатам. Допустим, что запрос был выдан пользователем MARTIN:

```
SQL> SELECT COUNT(1), AVG(sal) FROM hr.emp;
COUNT(SAL) AVG(SAL)
-----
14 1566.66667
```

Результат содержит данные 14 работников со средним окладом \$1566 — но на самом деле это средний оклад 6 работников, который разрешено видеть пользователю MARTIN, а не всех 14 работников. Иногда это создает путаницу. Если тот же запрос выдаст владелец схемы HR, результат будет другим:

```
SQL> CONN hr/hr
Connected.
SQL> SELECT COUNT(1), AVG(sal) FROM hr.emp;
COUNT(SAL) AVG(SAL)
-----
14 2073.21429
```

Так как результаты могут изменяться в зависимости от того, какой пользователь выдал запрос, будьте осторожны с интерпретацией результатов; в противном случае в вашем приложении могут появиться трудноуловимые ошибки.

Обратите внимание: показаны все 14 строк со всеми столбцами, но значения столбцов SAL и COMM в строках, которые пользователю видеть не положено (то есть работников отделов, отличных от 30), заменены на NULL.

Здесь RLS позволяет справиться с ситуациями, в которых требуется вывести все строки, но скрыть конфиденциальные значения. До выхода Oracle Database 10g того же результата можно было добиться при помощи представлений, но операции получались куда более сложными.

Отладка RLS

RLS — сложный механизм, зависящий от многих элементов архитектуры Oracle. При его использовании могут возникнуть ошибки, порожденные как недостатками архитектуры, так и неправильным применением со стороны пользователей. К счастью, для большинства ошибок RLS создает подробный файл трассировки в каталоге, заданном параметром инициализации базы данных USER_DUMP_DEST.

Интерпретация ошибок

Самая распространенная ошибка, с которой вы будете сталкиваться (и одновременно самая простая), — ORA-28110 (ошибка в функции политики или пакете). Исправление ошибок и перекомпиляция функции (или пакета, содержащего функцию) должны решить проблему.

Также могут встретиться ошибки времени выполнения — например, несоответствие типов или исключение VALUE_ERROR. В таких случаях Oracle иницирует ошибку ORA-28112 (ошибка при выполнении функции политики) и создает файл трассировки. Фрагмент этого файла выглядит так:

```
-----
Policy function execution error:
Logon user      : MARTIN
Table/View      : HR.EMP
Policy name     : EMP_DEPT_POLICY
Policy function: RLSOWNER.AUTHORIZED_EMPS
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at "RLSOWNER.AUTHORIZED_EMPS", line 14
ORA-06512: at line 1
```

Из данных трассировки видно, что при возникновении ошибки запрос выполнялся пользователем MARTIN. Здесь функция политики просто получила более одной строки. Просматривая код функции, вы замечаете, что она содержит сегмент следующего вида:

```
SELECT deptno
       INTO l_deptno
       FROM hr.emp
       WHERE ename = USER
```

Похоже, данные содержат более одного работника с именем MARTIN: количество выбранных строк превышает 1. Проблема решается либо обработкой ошибки через исключение, либо использованием другого условия в качестве предиката для получения номера отдела.

Исключение ORA-28113 (ошибка в предикате политики) возникает при некорректном построении предиката. Фрагмент файла трассировки с этой ошибкой:

```
Error information for ORA-28113:
Logon user      : MARTIN
Table/View      : HR.EMP
Policy name     : EMP_DEPT_POLICY
```

```
Policy function: RLSOWNER.AUTHORIZED_EMPS
RLS predicate :
DEPTNO = 10,
ORA-00907: missing right parenthesis
```

Из этого фрагмента видно, что функция политики возвращает следующий предикат:

```
DEPTNO = 10,
```

Условие синтаксически неверно, поэтому при попытке применения политики происходит сбой и запрос MARTIN не работает. Проблема решается исправлением логики функции политики и возвращением правильного предиката.

Выполнение прямых операций

Если вы используете прямые операции — прямую загрузку в SQL*Loader, прямую вставку с рекомендацией APPEND (INSERT /*+ APPEND */ INTO ...) или прямое экспортирование — учтите, что политики RLS для задействованных таблиц активизироваться *не будут*. В конце концов, прямые операции предназначены для работы в обход уровня SQL. Вам придется принять особые меры для таких ситуаций.

С экспортированием дело обстоит относительно просто. Вот что происходит при экспортировании таблицы EMP, защищенной одной или несколькими политиками RLS, с параметром DIRECT=Y:

```
About to export specified tables via Direct Path ...
EXP-00080: Data in table "EMP" is protected. Using conventional mode.
EXP-00079: Data in table "EMP" is protected. Conventional path may only be
exporting partial table.
```

Экспортирование выполняется успешно, но как видно из выходных данных, при выводе используется *традиционный*, а не прямой режим. А в процессе выполнения операции к таблице применяются политики RLS — то есть пользователь сможет экспортировать не все строки, а только те, которые ему разрешено видеть.



Из-за успешного завершения операции экспортирования таблицы с RLS может возникнуть ложное впечатление, будто были экспортированы все строки. Однако следует помнить, что экспортируются только те строки, которые пользователю разрешено видеть. Кроме того, хотя экспортирование вроде бы должно было выполняться в прямом режиме, оно выполняется в традиционном режиме.

Теперь при попытке выполнить прямую загрузку/вставку в SQL*Loader выдается сообщение об ошибке:

```
SQL> INSERT /*+ APPEND */
2 INTO hr.EMP
3 SELECT *
4 FROM hr.emp
5 WHERE rownum < 2;
from hr.emp
*
ERROR at line 4:
ORA-28113: policy predicate has error
```

Сообщение об ошибке неточно — в предикате на самом деле нет ошибки. Политика RLS не была применена из-за выполнения прямой операции, но в сообщении этот факт не отражен. Проблема решается либо временным отключением политики для таблицы EMP, либо экспортированием через учетную запись пользователя, обладающего системной привилегией EXEMPT ACCESS POLICY.

ОСНОВНЫЕ ПРИНЦИПЫ RLS

RLS автоматически применяет предикат (присоединяемый к предложению **WHERE**) ко всем запросам, выдаваемым пользователями; в результате часть строк таблицы может быть скрыта от пользователя.

- Предикат генерируется функцией политики, написанной пользователем.
- Политика таблицы определяет, выполнение каких условий должен обеспечивать предикат и какая функция политики должна выполняться.
- Для таблицы можно определить несколько политик RLS.
- Одна политика может применяться к нескольким таблицам.
- Тип политики (динамическая, статическая и т. д.) определяет частоту выполнения функции политики.
- Прямые операции загрузки таблицы выполняются в обход уровня SQL, политика RLS при этом не применяется, что приводит к ошибке.

Просмотр команд SQL

Во время отладки может возникнуть необходимость в просмотре команд, генерируемых Oracle при применении политики RLS, чтобы полностью избежать любых догадок и интерпретаций. Для просмотра замененных команд можно воспользоваться двумя способами.

Представления VPD

Первый способ — использование словарного представления **V\$VPD_POLICY**. VPD в имени означает «Virtual Private Database», то есть «виртуальная приватная база данных» — другое название для RLS. Представление содержит полную информацию о преобразовании запроса:

```
SQL> SELECT sql_text, predicate, POLICY, object_name
2   FROM v$sqlarea, v$vpd_policy
3  WHERE hash_value = sql_hash
```

| SQL_TEXT | PREDICATE |
|-----------------------------|-------------|
| POLICY | OBJECT_NAME |
| select count(*) from hr.emp | DEPTNO = 10 |
| EMP_DEPT_POLICY | EMP |

В столбце **SQL_TEXT** показана точная команда SQL, выданная пользователем, а в столбце **PREDICATE** — предикат, сгенерированный функцией политики и примененный к запросу. Используя это представление, можно идентифицировать команды, введенные пользователями, и примененные к ним предикаты.

Назначение события

Второй способ — назначение события в сеансе:

```
SQL> ALTER SESSION SET EVENTS '10730 trace name context forever, level 12';
```

Session altered.

```
SQL>SELECT COUNT(*) FROM hr.emp;
```

После завершения запроса в каталоге, заданном параметром инициализации базы данных `USER_DUMP_DEST`, появляется файл трассировки:

```
Logon user      : MARTIN
Table/View     : HR.EMP
Policy name    : EMP_DEPT_POLICY
Policy function: RLSOWNER.AUTHORIZED_EMPS
RLS view :
SELECT  "EMPNO", "ENAME", "JOB", "MGR", "HIREDATE", "SAL", "COMM", "DEPTNO"
FROM    "HR"."EMP" "EMP" WHERE (DEPTNO = 10)
```

Любой из этих двух способов позволит вам увидеть точную команду, полученную в результате замены пользовательского запроса.

Контексты приложений

При обсуждении безопасности уровня строк в предыдущем разделе было сделано очень важное допущение: предикат (то есть условие, ограничивающее набор видимых строк таблицы) оставался неизменным. В рассмотренных примерах он базировался на коде отдела пользователя. Допустим, в системе вводится новое требование: теперь пользователи просматривают записи работников на основании не кодов отделов, а специально ведущихся для этой цели списков привилегий. В таблице `EMP_ACCESS` хранится информация о том, кому из пользователей разрешено работать с теми или иными данными.

```
SQL> DESC emp_access
Name          Null?     Type
-----
USERNAME      NUMBER
DEPTNO        NUMBER
```

Примерный вид данных:

| USERNAME | DEPTNO |
|----------|--------|
| MARTIN | 10 |
| MARTIN | 20 |
| KING | 20 |
| KING | 10 |
| KING | 30 |
| KING | 40 |

Пользователь `Martin` может просматривать данные отделов с кодами 10 и 20, а пользователь `King` — данные отделов 10, 20, 30 и 40. Если имя пользователя не указано в таблице, он не может просматривать записи. Новое правило требует, чтобы предикаты генерировались динамически в функции политики.

Кроме того, должна быть предусмотрена возможность динамического изменения привилегий пользователей посредством обновления таблицы `EMP_ACCESS`, и пользователя при этом не следует заставлять заново подключаться к базе данных. Следовательно, триггер `LOGON` (см. главу 19) нам не поможет.

Одно из возможных решений заключается в создании пакета с переменной, в которой хранится предикат; пользователю предоставляется возможность выполнения сегмента кода PL/SQL, присваивающего значение переменной. Внутри функции политики значение пакетной переменной используется в качестве предиката. Насколько приемлемо это решение? Подумайте хорошенько: если пользователь может присвоить другое значение пакетной переменной, что помешает ему присвоить привилегированное значение? Пользователь подключается к базе данных, присваивает переменной значение, открывающее доступ ко всем данным, после чего выполняет выборку и видит все записи. Отсутствие безопасности делает этот вариант неприемлемым. Собственно, этот сценарий наглядно

демонстрирует, почему код задания значений переменных следует размещать в триггере LOGON, где пользователь не сможет внести изменения.

Использование контекстов приложений

Вероятность того, что пользователь может динамически изменить пакетную переменную, заставляет переосмыслить стратегию. Нам необходим механизм задания глобальной переменной неким безопасным механизмом, исключающим несанкционированные изменения. К счастью, Oracle предоставляет такую возможность. *Контекст приложения* аналогичен глобальной пакетной переменной; после задания значения он остается доступным на протяжении всего сеанса. Впрочем, на этом сходство кончается. Важнейшее различие заключается в том, что в отличие от пакетной переменной, контекст приложения не задается простым присваиванием; для изменения значения необходим вызов процедуры — и это обстоятельство делает этот вариант более безопасным.

Как и структуры языка С или записи PL/SQL, контекст приложения обладает атрибутами, которым присваиваются значения. Однако в отличие от аналогов из С и PL/SQL, имена атрибутов не фиксируются при создании контекста; это происходит во время выполнения. Контексты приложений по умолчанию хранятся в области PGA, если только они не определены как глобальные. Так как область PGA содержит приватные данные сеанса, хранящиеся в ней значения остаются невидимыми для других сеансов.

В следующем примере команда CREATE CONTEXT используется для определения нового контекста с именем dept_ctx:

```
SQL> CREATE CONTEXT dept_ctx USING set_dept_ctx;
```

Context created.

Секция USING set_dept_ctx означает, что существует процедура с именем set_dept_ctx, и только эта процедура может изменять атрибуты контекста dept_ctx. Никаким другим способом атрибуты изменяться не могут. У созданного контекста еще нет атрибутов — пока мы просто определили общий контекст (имя и безопасный механизм изменения). На следующем шаге необходимо создать процедуру, в которой атрибутам контекста будут присваиваться значения с использованием функции SET_CONTEXT встроенного пакета DBMS_SESSION, как показано в следующем примере:

```
PROCEDURE set_dept_ctx (p_attr IN VARCHAR2, p_val IN VARCHAR2)
IS
BEGIN
  DBMS_SESSION.set_context ('DEPT_CTX', p_attr, p_val);
END;
```

Чтобы присвоить атрибуту DEPTNO значение 10, мы выполняем следующую команду:

```
SQL> EXEC set_dept_ctx ('DEPTNO', '10')
```

PL/SQL procedure successfully completed.

Для получения текущего значения атрибута вызывается функция SYS_CONTEXT, которая получает два параметра: имя контекста и имя атрибута. Пример:

```
SQL> DECLARE
2   l_ret   VARCHAR2 (20);
3   BEGIN
4     l_ret := SYS_CONTEXT ('DEPT_CTX', 'DEPTNO');
5     DBMS_OUTPUT.put_line ('Value of DEPTNO = ' || l_ret);
6   END;
7   /
```

Value of DEPTNO = 10

Функция также может использоваться для получения некоторых predefined контекстов, например IP-адресов и терминалов клиентов:

```
BEGIN
    DBMS_OUTPUT.put_line ( 'The Terminal ID is '
                          || SYS_CONTEXT ('USERENV', 'TERMINAL')
                          );
END;
```

Результат:

The Terminal ID is pts/0

В этом фрагменте используется predefined контекст `USERENV`, обладающий такими атрибутами, как `TERMINAL`, `IP_ADDRESS`, `OS_USER` и т. д. Значения этих атрибутов присваиваются автоматически, и изменить их в приложении невозможно — допускается только чтение данных.

Безопасность в контекстах

В сущности, работа процедуры `set_dept_ctx` сводится к вызову готовой программы `DBMS_SESSION.SET_CONTEXT` с соответствующими параметрами. Зачем определять для этого процедуру? Почему бы не вызвать встроенную функцию напрямую? Давайте посмотрим, что произойдет, если пользователь попытается в том же сегменте кода присвоить значение атрибуту `DEPTNO`:

```
SQL> BEGIN
2     DBMS_SESSION.set_context
3     ('DEPT_CTX', 'DEPTNO',10);
4 END;
5 /
begin
*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 82
ORA-06512: at line 2
```

Ошибка `ORA-01031` выглядит довольно странно: пользователь `Martin` обладает привилегией `EXECUTE` для `DBMS_SESSION`, так что проблема, очевидно, не в нехватке привилегий. Вы можете убедиться в этом, заново предоставив привилегию `EXECUTE` для пакета и повторно выполнив тот же сегмент кода; вы получите ту же ошибку.

Дело в том, что контексты приложений не могут изменяться прямыми обращениями к встроенному пакету. Все модификации должны осуществляться программным блоком, связанным с контекстом при его создании. Такой блок называется *доверенной программой* контекста приложения.



При создании контекста приложения необходимо указать его доверенную программу. Только доверенная программа может задавать значения в этом контексте, но не в других контекстах.

Контексты как предикаты в RLS

Итак, для изменения значений атрибутов контекста должна использоваться процедура. Не приводит ли это к бессмысленному увеличению сложности программы?

Нет, потому что доверенная процедура — *единственный* механизм изменения контекста — играет роль «стража» для работы с контекстом. В ней могут выполняться сколь угодно

сложные действия по аутентификации и проверке данных, гарантирующие действительность присваивания. Мы даже можем полностью отказаться от передачи параметров и задавать их на основании предопределенных значений без получения данных от пользователя. Например, из поставленных требований известно, что контексту должна быть присвоена строка с кодами отделов, прочитанными из таблицы EMP_ACCESS (а не передаваемыми пользователем!). Затем контекст приложения используется в функции политики. Давайте посмотрим, как реализовать это требование.

Сначала необходимо внести изменения в функцию политики:

```
/* Файл в Сети: authorized_emps_3.sql */
1  FUNCTION authorized_emps (
2      p_schema_name  IN  VARCHAR2,
3      p_object_name  IN  VARCHAR2
4  )
5      RETURN VARCHAR2
6  IS
7      l_deptno        NUMBER;
8      l_return_val    VARCHAR2 (2000);
9  BEGIN
10     IF (p_schema_name = USER)
11     THEN
12         l_return_val := NULL;
13     ELSE
14         l_return_val := SYS_CONTEXT ('DEPT_CTX', 'DEPTNO_LIST');
15     END IF;
16
17     RETURN l_return_val;
18 END;
```

Функция политики предполагает, что коды отделов будут передаваться через атрибут DEPTNO_LIST контекста dept_ctx (строка 14). Чтобы задать значение атрибута, необходимо внести изменения в доверенную процедуру контекста:

```
/* Файл в Сети: set_dept_ctx_2.sql */
1  PROCEDURE set_dept_ctx
2  IS
3      l_str  VARCHAR2 (2000);
4      l_ret  VARCHAR2 (2000);
5  BEGIN
6      FOR deptrec IN (SELECT deptno
7                      FROM emp_access
8                      WHERE username = USER)
9      LOOP
10         l_str := l_str || deptrec.deptno || ',';
11     END LOOP;
12
13     IF l_str IS NULL
14     THEN
15         -- Информация о доступе не найдена, строки должны
16         -- остаться невидимыми для пользователя.
17         l_ret := '1=2';
18     ELSE
19         l_str := RTRIM (l_str, ',');
20         l_ret := 'DEPTNO IN (' || l_str || ')';
21         DBMS_SESSION.set_context ('DEPT_CTX', 'DEPTNO_LIST', l_ret);
22     END IF;
23 END;
```

Пора протестировать функцию. Сначала пользователь Martin подключается к базе данных и подсчитывает количество работников. Перед выдачей запроса он должен установить контекст:

```
SQL> EXEC rlsowner.set_dept_ctx
```

PL/SQL procedure successfully completed.

```
SQL> SELECT SYS_CONTEXT ('DEPT_CTX', 'DEPTNO_LIST') FROM DUAL;
```

```
SYS_CONTEXT('DEPT_CTX', 'DEPTNO_LIST')
-----
DEPTNO IN (20,10)
```

```
SQL> SELECT DISTINCT deptno FROM hr.emp;
```

```
DEPTNO
-----
      10
      20
```

В соответствии с таблицей EMP_ACCESS ему видны только данные работников отделов 10 и 20. Допустим, в данных Martin номер отдела меняется на 30. Администратор вносит соответствующие изменения в таблицу:

```
SQL> DELETE emp_access WHERE username = 'MARTIN';
```

2 rows deleted.

```
SQL> INSERT INTO emp_access VALUES ('MARTIN',30);
```

1 row created.

```
SQL> COMMIT;
```

Commit complete.

Теперь при выполнении тех же запросов Martin получит другой результат:

```
SQL> EXEC rlsowner.set_dept_ctx
```

PL/SQL procedure successfully completed.

```
SQL> SELECT SYS_CONTEXT ('DEPT_CTX', 'DEPTNO_LIST') FROM DUAL;
```

```
SYS_CONTEXT('DEPT_CTX', 'DEPTNO_LIST')
-----
DEPTNO IN (30)
```

```
SQL> SELECT DISTINCT deptno FROM hr.emp;
```

```
DEPTNO
-----
      30
```

Изменения вступают в силу автоматически. Поскольку Martin не задает атрибуты контекста вручную, такое решение по своей природе более безопасно, чем задание глобальной переменной. Кроме того, контекстная политика RLS в Oracle10g и более поздних версиях также способствует повышению производительности. Функция политики выполняется только при изменении контекста, а между изменениями используются кэшированные значения. Тем самым обеспечивается более высокая скорость работы этой политики по сравнению с используемой по умолчанию динамической политикой. Чтобы определить политику как контекстную, следует передать процедуре DBMS_RLS.add_policy дополнительный параметр:

```
policy_type => DBMS_RLS.context_sensitive
```


Итак, чем же этот метод отличается от создания динамически сгенерированной функции политики для таблицы `emp_access`? В случае функции политики необходимо ее выполнение для получения значения предиката (списка отделов в нашей ситуации). Предположим, имеется таблица, к которой обращены миллионы запросов; функция политики будет выполняться многократно, при этом каждый раз будет происходить обращение к таблице `emp_access` — с печальными последствиями для производительности. Политику можно определить как статическую, чтобы функция не выполнялась так часто, но тогда при изменении записей `emp_access` функция политики не будет учитывать изменения и выдаст неправильный результат. Определение контекстной политики в контексте приложения решает обе проблемы — функция политики будет выполняться заново при изменении контекстного значения. Контекстные значения хранятся в памяти, поэтому обращения к ним происходят очень быстро.

В Oracle Database 12c контекстная политика предоставляет еще больше преимуществ. Кроме параметра `policy_type`, можно передать еще два параметра — `namespace` (пространство имен, а проще говоря — имя) и `attribute` (атрибут контекста, изменение которого должно приводить к повторному выполнению функции политики). В приведенном выше примере я использовал контекст `DEPT_CTX`, а конкретнее — атрибут с именем `DEPTNO_LIST`, для вывода списка номеров отделов, которые разрешено видеть пользователю. Функцию политики можно определить так, чтобы она повторно выполнялась только при изменении списка отделов. Для этого в политику добавляются зависимости от контекста и атрибута:

```
BEGIN
  DBMS_RLS.alter_policy (object_schema => 'HR',
                        object_name    => 'EMP',
                        policy_name    => 'EMP_POLICY',
                        alter_option   => DBMS_RLS.add_attribute_association,
                        namespace      => 'DEPTNO_CTX',
                        attribute       => 'DEPTNO_LIST');
END;
```

Пример полного вызова процедуры `add_policy`:

```
BEGIN
  DBMS_RLS.add_policy (object_schema    => 'HR',
                      object_name       => 'EMP',
                      policy_name       => 'EMP_POLICY',
                      function_schema   => 'RLSOWNER',
                      policy_function   => 'AUTHORIZED_EMPS',
                      statement_types    => 'SELECT',
                      update_check      => TRUE,
                      sec_relevant_cols => 'SAL, COMM',
                      sec_relevant_cols_opt => DBMS_RLS.all_rows,
                      policy_type       => DBMS_RLS.context_sensitive,
                      namespace         => 'DEPT_CTX',
                      attribute          => 'DEPTNO_LIST'
                      );
END;
```

Идентификация сторонних пользователей

Полезность контекстов приложений выходит далеко за рамки ситуаций, описанных выше. Самое важное применение контекстов приложения — возможность различать пользователей, которые не могут быть идентифицированы на уровне уникального сеанса. Этот сценарий типичен для веб-приложений, использующих пул подключений с одним пользователем (например, `CONNPOOL`). Веб-пользователи подключаются к серверу приложения, который в свою очередь использует одно из подключений пула для обращения к базе данных (рис. 23.6).

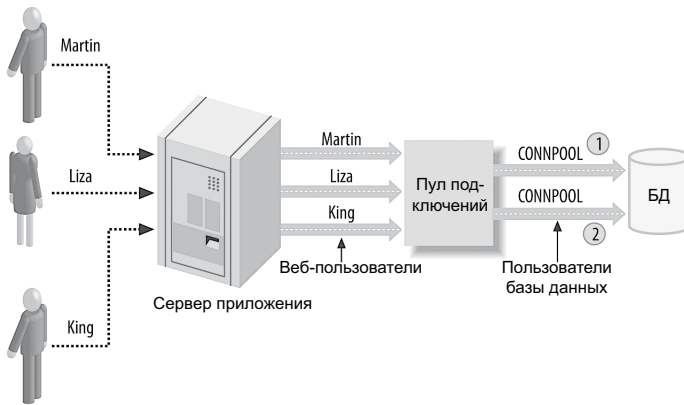


Рис. 23.6. Пользователи приложения и RLS

В этой схеме пользователи **Martin** и **King** не являются пользователями базы данных; это веб-пользователи, и базе данных о них ничего не известно. Пул подключений связывается с базой данных с идентификатором **CONNPOOL**, который является пользователем базы данных. Когда **Martin** запрашивает информацию из базы данных, пул должен решить использовать для выборки подключение 1. После завершения запроса подключение освобождается. Если в этот момент пользователь **King** запросит данные, пул может решить использовать то же подключение (1). Соответственно, с точки зрения базы данных сеанс (а на самом деле подключение из пула) принадлежит пользователю **CONNPOOL**. По этой причине в приведенных ранее примерах (в которых использовалась функция **USER**) идентификация пользователя работать не будет. Функция **USER** всегда возвращает **CONNPOOL**, потому что к базе данных подключен именно этот пользователь.

На помощь приходят контексты приложения. Предположим, имеется контекст с именем **WEB_CTX** и атрибутом **WEBUSER**. Клиент присваивает ему имя фактического пользователя (например, **MARTIN**) при отправке запроса к пулу подключений:

```
BEGIN
    set_web_ctx ('WEBUSER', 'MARTIN');
END;
```

Политика RLS может базироваться на этом значении вместо имени пользователя базы данных. В таком случае функция политики будет выглядеть немного иначе:

```
/* Файл в Сети: authorized_emps_4.sql */
1  FUNCTION authorized_emps (
2      p_schema_name  IN   VARCHAR2,
3      p_object_name  IN   VARCHAR2
4  )
5      RETURN VARCHAR2
6  IS
7      l_deptno       NUMBER;
8      l_return_val   VARCHAR2 (2000);
9  BEGIN
10     IF (p_schema_name = USER)
11     THEN
12         l_return_val := NULL;
13     ELSE
14         SELECT deptno
15             INTO l_deptno
16             FROM emp
17             WHERE ename = SYS_CONTEXT ('WEB_CTX', 'WEBUSER');
18
19         l_return_val := 'DEPTNO = ' || l_deptno;
20     END IF;
```

```
21
22     RETURN l_return_val;
23 END;
```

Обратите внимание на строку 17. В исходной версии кода она выглядела так:

```
WHERE ename = USER;
```

А теперь выглядит так:

```
WHERE ename = SYS_CONTEXT ('WEB_CTX', 'WEBUSER');
```

Это условие получает имя веб-пользователя и сравнивает его со столбцом ENAME.

ОСНОВНЫЕ ПРИНЦИПЫ КОНТЕКСТОВ ПРИЛОЖЕНИЙ

- Контексты напоминают глобальные пакетные переменные; после присваивания они сохраняют свои значения, к которым можно обращаться на протяжении сеанса. Однако в каждом сеансе значения переменных могут задаваться по-разному. Контексты хранятся в PGA.
- Контексты могут иметь один и более именованных атрибутов, каждому из которых присваивается некоторое значение. Атрибуты определяются во время выполнения.
- Чтобы задать значение атрибута, следует вызвать его доверенную процедуру, указанную в команде CREATE CONTEXT. Единственными аргументами доверенной процедуры являются имя и значение атрибута.
- После того как контекст будет определен, для его получения можно воспользоваться функцией SYS_CONTEXT.

Детализированный аудит

Детализированный аудит (FGA, Fine-Grained Auditing) предоставляет механизм для регистрации факта выдачи пользователями определенных команд и выполнения некоторых условий. При этом регистрируется команда, введенная пользователем, а также другая информация: время, терминал и т. д.

Под *традиционным аудитом* в Oracle понимается механизм регистрации того, какая схема выполнила то или иное действие: пользователь *Joe* выполняет процедуру *X*, пользователь *John* выполнил выборку данных из таблицы *Y* и т. д. Протокол всех этих действий, называемый *журналом аудита*, хранится в таблице *AUD\$* в схеме *SYS*, доступной для всех пользователей через несколько представлений словаря данных — например, *DBA_AUDIT_TRAIL*. Журналы аудита также могут записываться в файлы операционной системы вместо таблиц базы данных. Независимо от того, где хранится эта информация, главный недостаток традиционного аудита остается неизменным: он следит за тем, кто выполнил ту или иную команду, а не за тем, что при этом было сделано. Например, из журнала аудита можно узнать, что пользователь *Joe* получил данные из таблицы *ACCOUNTS*, но какие именно записи — остается неизвестным. Если вы хотите знать измененные значения, приходится устанавливать для таблиц триггеры DML (см. главу 19) и сохранять значения в таблице, определенной вами. Но поскольку определить триггер для команды *SELECT* не удастся, для ситуации с *Joe* этот вариант не подойдет.

На помощь приходит механизм FGA. Функциональность FGA используется через встроенный пакет *DBMS_FGA*. Механизм FGA, появившийся в Oracle9i Database, изначально применялся только к командам *SELECT*; начиная с Oracle Database 10g он применяется ко всем командам DML.



Не путайте FGA с FGAC (Fine Grained Access Control) — этот термин является синонимом RLS (эта тема рассматривалась ранее в этой главе).

При помощи FGA можно сохранить описания операций `SELECT`, `INSERT`, `UPDATE` и `DELETE` в журнале аудита (хотя и в другом журнале, не в таблице `AUD$`). Из журнала вы получите информацию не только о том, кто выполнил ту или иную операцию, но и множество других полезных сведений — точную команду, введенную пользователем, код `SCN` (System Change Number), значения подставляемых параметров (если они использовались) и многое другое.

Одна из самых полезных особенностей FGA — возможность избирательного применения для конкретных операций. Например, аудит может выполняться при выборке из столбца `SAL`, но не из других столбцов; или операция может регистрироваться в журнале аудита только при выборке из столбца `SAL`, и если значение этого столбца не менее 1500. Избирательная регистрация сокращает объем генерируемых данных аудита.

Другая чрезвычайно полезная особенность FGA связана с возможностью автоматического выполнения процедуры, определяемой пользователем. Некоторые практические применения этой возможности описаны в следующих разделах.

Зачем изучать FGA?

FGA — «близкий родственник» традиционного аудита. Безусловно, эта возможность предназначена в первую очередь для администраторов баз данных; зачем же разработчикам PL/SQL изучать ее? Есть несколько причин:

Безопасность. Как уже упоминалось для других возможностей, умение пользоваться встроенными средствами безопасности Oracle в наши дни попросту является одной из составляющих качественного проектирования. Средства FGA должны стать частью вашего инструментария.

Производительность. Другая, более убедительная причина — практическая ценность информации, возвращаемой FGA. Кроме информации о том, кто выполнил ту или иную команду, FGA может точно описать выполненные команды SQL. Если включить FGA для таблицы, то все команды SQL, обращенные к этой таблице, будут сохраняться в журналах аудита FGA. Последующий анализ этой информации поможет выявить закономерности в выполнении команд и решить, не нужно ли добавить или изменить индексы или внести другие изменения, способствующие повышению производительности.

Подставляемые параметры. FGA также сохраняет значения параметров, а в любом нормально спроектированном приложении используется множество подставляемых параметров. Как узнать, какие значения передавались во время запуска приложения? На основании собранной информации вы решите, нужно ли определять дополнительный индекс. Журналы FGA предоставляют сведения, упрощающие принятие таких решений.

Модули-обработчики. FGA также может выполнить процедуру, называемую *модулем-обработчиком*, при выполнении некоторых условий аудита. Скажем, при включении FGA для команд `SELECT` модуль-обработчик будет выполняться для каждой команды `SELECT`, обращенной к таблице. Происходящее напоминает включение триггера для команды `SELECT` — Oracle не поддерживает эту возможность, но она может оказаться в высшей степени полезной. Предположим, каждый раз, когда пользователь запрашивает информацию об окладе руководства компании, сообщение должно ставиться в расширенную очередь для последующей передачи в другую базу данных. Для решения этой задачи

можно реализовать модуль-обработчик, который будет делать то же самое, что делает триггер для SELECT.

Давайте поближе познакомимся с применением FGA в ваших приложениях.



Для правильной работы FGA база данных должна работать в режиме затратного оптимизатора (CBO), запросы должны использовать CBO (то есть они не должны содержать рекомендации RULE), а таблицы (или представления) в запросе должны быть проанализированы (по крайней мере с оценками). Если эти условия не выполняются, при использовании FGA возможны ложноположительные срабатывания: информация будет записываться в журнал аудита, хотя содержимое столбца на самом деле не выбирается.

Простой пример

Начнем с простого примера — таблица EMP из схемы HR уже знакома вам по обсуждению RLS в предыдущем разделе. Предположим, для контроля за использованием конфиденциальной информации вы хотите регистрировать в журнале все операции выборки столбцов SAL и COMM. Чтобы сократить объем генерируемого журнала, регистрироваться будут только операции со строками, в которых оклад составляет \$150 000 и более. Наконец, аудит также должен включаться при запросе оклада работника 100 (то есть вас). Разобравшись с требованиями, перейдем к построению архитектуры FGA. Как и в случае с RLS, все начинается с определения политики FGA для таблицы. Политика определяет все условия, в которых должен срабатывать механизм аудита. Для добавления политики используется процедура ADD_POLICY из встроенного пакета DBMS_FGA:

```
1 BEGIN
2     DBMS_FGA.add_policy (object_schema      => 'HR',
3                          object_name        => 'EMP',
4                          policy_name        => 'EMP_SEL',
5                          audit_column       => 'SAL, COMM',
6                          audit_condition    => 'SAL >= 150000 OR EMPID = 100'
7                          );
8 END;
```

Я определяю политику FGA с именем EMP_SEL (строка 4), передавая ее в параметре policy_name. Политика определяется для таблицы EMP (строка 3), владельцем которой является HR (строка 2).

Политика требует сохранять данные в журнале аудита каждый раз, когда пользователь выбирает данные из двух столбцов, SAL и COMM (строка 5). Однако данные сохраняются только в том случае, если значение SAL в этой строке не менее \$150 000 или если идентификатор работника равен 100 (условие аудита, строка 6).

Параметры audit_column и audit_condition не являются обязательными. Если они не указаны, то в аудите будет участвовать каждая команда SELECT к таблице EMP схемы HR. Начиная с Oracle Database 10g, поскольку FGA также может применяться и к обычным операциям DML, я могу определять конкретные команды, на которые должна распространяться политика аудита, при помощи нового параметра statement_types:

```
1 BEGIN
2     DBMS_FGA.add_policy (object_schema      => 'HR',
3                          object_name        => 'EMP',
4                          policy_name        => 'EMP_DML',
5                          audit_column       => 'SALARY, COMM',
6                          audit_condition    => 'SALARY >= 150000 OR EMPID = 100',
7                          statement_types    => 'SELECT, INSERT, DELETE, UPDATE'
8                          );
9 END;
```



Хотя политика используется и в FGA, и в RLS, она играет совершенно разные роли. Впрочем, имеется и сходство — политика FGA, как и ее «родственник» в RLS, не является «объектом схемы», то есть не принадлежит никакому пользователю. Каждый пользователь, обладающий привилегией EXECUTE для пакета DBMS_FGA, может создавать политики и удалять политики, созданные другим пользователем. Ваш администратор базы данных должен быть очень осмотрительным при предоставлении привилегий EXECUTE для этого встроенного пакета; предоставление привилегии категории PUBLIC делает всю вашу деятельность по аудиту сомнительной — в лучшем случае.

В нашем примере журнал аудита записывается только в том случае, если:

- Пользователь получает данные из одного или обоих столбцов, SAL и COMM.
- Значение SAL не менее 150 000 или значение EMPID равно 100.

Для сохранения информации в журнале аудита истинными должны быть оба условия. Если одно условие истинно, а другое нет, то операция не регистрируется. Если пользователь в своем запросе не получает либо столбец SAL, либо столбец COMM (прямо или косвенно), то запись в журнале не генерируется, даже если у записи столбец SAL содержит значение 150 000 и более. Предположим, оклад Джейка равен 160 000, а его идентификатор EMPID равен 52. Пользователь, который просто хочет узнать имя руководителя, выдает следующий запрос:

```
SELECT mgr
  FROM emp
 WHERE empid = 52;
```

Так как в запросе ни столбец SAL, ни столбец COMM не задействован, операция не регистрируется. Однако запрос:

```
SELECT mgr
  FROM emp
 WHERE sal >= 160000;
```

будет зарегистрирован в журнале. Почему? Потому что столбец SAL задействован в секции WHERE, то есть пользователь неявно обращается к нему; следовательно, условие аудита выполнено. Кроме того, значение SAL прочитанных записей больше 150 000. Так как оба условия выполнены, выполняется аудит.

Условие аудита не обязано ссылаться на столбцы таблицы, для которой определена политика; оно также может ссылаться на другие значения (например, псевдостолбцы). Это может быть полезно при включении аудита для некоторого подмножества пользователей. Допустим, вы хотите регистрировать обращения к таблице EMP от пользователя Scott. Политика определяется следующим образом:

```
BEGIN
  DBMS_FGA.add_policy (object_schema    => 'HR',
                       object_name      => 'EMP',
                       policy_name      => 'EMP_SEL',
                       audit_column     => 'SALARY, COMM',
                       audit_condition  => 'USER='SCOTT'');
END;
```

Количество столбцов

В предыдущем примере список столбцов был задан следующим образом:

```
audit_column    => 'SAL, COMM'
```

Это означает, что регистрируются все обращения пользователя к столбцу SAL или COMM. Однако в некоторых ситуациях могут действовать более точные требования: обращение

ко *всем* перечисленным столбцам, а не только к одному из них. Например, в базе данных `employee` механизм FGA должен регистрировать только обращения к `SAL` и `EMPNAME` одновременно. Если в запросе задействован только один столбец, операция вряд ли раскроет конфиденциальную информацию. Выполняется следующий запрос:

```
SELECT salary FROM hr.emp;
```

Он выводит оклады всех работников, но без указания имен рядом с окладами эта информация окажется бесполезной для пользователя, желающего узнать оклад конкретного работника. Аналогичным образом запрос:

```
SELECT empname
FROM hr.emp;
```

вернет имена работников без окладов, то есть конфиденциальность останется под защитой. Однако при выдаче запроса

```
SELECT empname, salary FROM hr.emp;
```

пользователь узнает, сколько получает каждый из работников — это та самая информация, которую следует защитить.

Из трех показанных случаев запись в журнал аудита должна происходить только для последнего (единственного, для которого запрос вернет содержательную информацию). В Oracle9i Database не было возможности задать комбинацию столбцов как условие аудита; в Oracle Database 10g и выше это стало возможно благодаря параметру `audit_column_opts` процедуры `ADD_POLICY`. По умолчанию значение параметра равно `DBMS_FGA.ANY_COLUMNS`, то есть запись в журнал аудита производится при обращении к любому из столбцов. Если же задать параметру значение `DBMS_FGA.ALL_COLUMNS`, то данные аудита генерируются только при обращении ко всем перечисленным столбцам. В моем примере, чтобы политика FGA создавала запись аудита только при выборке столбцов `SALARY` и `EMPNAME`, она должна выглядеть так:

```
BEGIN
  DBMS_FGA.add_policy (object_schema      => 'HR',
                      object_name         => 'EMP',
                      policy_name         => 'EMP_DML',
                      audit_column        => 'SALARY, EMPNAME',
                      audit_condition     => 'USER='SCOTT'',
                      statement_types     => 'SELECT, INSERT, DELETE, UPDATE',
                      audit_column_opts   => DBMS_FGA.all_columns
                      );
END;
```

Данная возможность чрезвычайно полезна — она позволяет ограничиться ведением журнала аудита для актуальных операций, чтобы сократить объем данных журнала.

Просмотр журнала аудита

Данные аудита FGA записываются в таблицу `FGA_LOG$`, принадлежащую схеме `SYS`. Для работы с данными аудита используется внешний интерфейс — представление словаря данных `DBA_FGA_AUDIT_TRAIL`:

```
SELECT db_user, sql_text
FROM dba_fga_audit_trail
WHERE object_schema = 'HR' AND object_name = 'EMP'
```

Запрос выдает следующий результат:

```
DB_USER SQL_TEXT
-----
SCOTT   select salary from hr.emp where empid = 1
```

Кроме пользователя и команды SQL, в журнале FGA также сохраняются другие полезные сведения. Некоторые важные столбцы представления:

- **TIMESTAMP** — время выполнения операции.
- **SCN** — номер изменения системы при выполнении операции. Для просмотра прошлых SCN используются *ретроспективные запросы* Oracle.
- **OS_USER** — пользователь операционной системы, подключенный к базе данных.
- **USERHOST** — терминал или клиентский компьютер, с которого подключается пользователь.
- **EXT_NAME** — если пользователь проходит внешнюю аутентификацию (например, средствами LDAP), переданное такому внешнему механизму аутентификации имя становится важным; оно сохраняется в этом столбце.

ОСНОВНЫЕ ПРИНЦИПЫ FGA

- FGA может сохранять информацию обо всех обращениях к таблице из команд SELECT (в Oracle9i Database) или любых типов операций DML (в Oracle Database 10g и выше) в таблице аудита с именем FGA_LOG\$ схемы SYS.
- Генерирование данных журнала аудита можно ограничить таким образом, чтобы информация записывалась в журнал только при выборке некоторых столбцов или выполнении некоторых условий.
- Для правильной работы FGA должен использоваться затратный оптимизатор; в противном случае возможны ложноположительные срабатывания.
- Для записи в журнал используется автономная транзакция. Таким образом, если при выполнении операции DML произойдет ошибка, запись все равно появится — что также может приводить к ложноположительным срабатываниям.
- В журнале аудита сохраняется точная команда, введенная пользователем, значения подставляемых параметров (если есть), номер изменения системы на момент выполнения запроса, а также различные атрибуты сеанса: имя пользователя базы данных, имя пользователя операционной системы, временная метка и т. д.
- Кроме записи данных в журнал аудита, FGA также может автоматически выполнить процедуру, называемую модулем-обработчиком.

Подставляемые параметры

Описывая FGA, я упомянул о подставляемых параметрах. Давайте посмотрим, как использовать FGA с этими переменными. Предположим, вместо «обычной» команды в форме:

```
SELECT salary
  FROM emp
 WHERE empid = 100;
```

был использован блок следующего вида:

```
DECLARE
  l_empid PLS_INTEGER;
BEGIN
  SELECT salary
    FROM emp
   WHERE empid = l_empid;
END;
```

FGA сохраняет значения подставляемых параметров вместе с текстом команды SQL. Сохраненные значения находятся в столбце SQL_BIND представления DBA_FGA_AUDIT_TRAIL. В предыдущем примере используется код:


```
SQL> SELECT sql_text,sql_bind from dba_fga_audit_trail;
```

| SQL_TEXT | SQL_BIND |
|---|-----------|
| select * from hr.emp where empid = :empid | #1(3):100 |

Обратите внимание на формат вывода сохраненного параметра:

#1(3):100

Здесь #1 — номер параметра. Если запрос использует более одного подставляемого параметра, то другим будут присвоены номера #2, #3 и т. д. Фрагмент(3) обозначает фактическую длину переменной. В нашем примере пользователь Scott указал значение 100, поэтому длина равна 3. Фрагмент :100 обозначает фактическое значение параметра (100).

Если в запросе используется несколько подставляемых параметров, столбец SQL_BIND содержит строку значений. Например, для запроса:

```
DECLARE
  l_empid   PLS_INTEGER := 100;
  l_salary  NUMBER := 150000;

  TYPE emps_t IS TABLE OF emp%ROWTYPE;
  l_emps     emps_t;
BEGIN
  SELECT * BULK COLLECT INTO l_emps
    FROM hr.emp
   WHERE empid = l_empid OR salary > l_salary;
END;
```

столбец SQL_BIND будет выглядеть так:

#1(3):100 #2(5):150000



Текст команды SQL и подставляемые параметры сохраняются только в том случае, если параметру audit_trail процедуры ADD_POLICY задано значение DB_EXTENDED (по умолчанию), но не значение DB.

Модули-обработчики

Как упоминалось ранее, механизм FGA также позволяет выполнять хранимые программные блоки PL/SQL — например, хранимые процедуры. Если хранимая процедура, в свою очередь, содержит внешнюю программу, эта программа тоже будет выполнена. Хранимая программа называется *модулем-обработчиком* (handler module). В приведенном ранее примере с построением механизма для аудита обращений к таблице EMP я также мог бы указать хранимую процедуру (отдельную или пакетную), которая должна выполняться при аудите. Если владельцем хранимой процедуры является пользователь FGA_ADMIN и она называется myproc, то процедура создания политики ADD_POLICY будет вызываться с двумя новыми параметрами — handler_schema и handler_module:

```
BEGIN
  DBMS_FGA.add_policy (object_schema => 'HR',
    object_name      => 'EMP',
    policy_name      => 'EMP_SEL',
    audit_column     => 'SALARY, COMM',
    audit_condition  => 'SALARY >= 150000 OR EMPID = 100',
    handler_schema   => 'FGA_ADMIN',
    handler_module   => 'MYPROC'
  );
END;
```

При выполнении условий аудита и обращениях к соответствующим столбцам помимо регистрации операции в журнале также выполняется процедура `fga_admin.myproc`. Она выполняется автоматически при каждой записи данных аудита в виде *автономной транзакции* (см. главу 14). Процедура получает ровно три параметра — имя схемы, имя таблицы и имя политики. Структура процедуры модуля-обработчика:

```
PROCEDURE myproc (  
    p_table_owner  IN  VARCHAR2,  
    p_table_name   IN  VARCHAR2,  
    p_fga_policy   IN  VARCHAR2  
)  
IS  
BEGIN  
    -- здесь размещается код  
END;
```

Как использовать эту возможность? Есть множество вариантов. Например, вы можете построить собственную программу, которая будет сохранять данные в ваших таблицах наряду с записью в стандартные таблицы журналов аудита. Можно записывать сообщения в очереди для размещения в других базах данных, запросить отправку электронной почты администраторам по безопасности, просто подсчитать количество выполнений условий аудита... Словом, возможности безграничны.



Если при выполнении модуля-обработчика происходит какая-либо ошибка, FGA не сообщает об ошибке при запросе данных из таблицы. Вместо этого FGA просто прекращает выборку строк, для которых выполнение модуля-обработчика завершилось неудачей. Это коварная ситуация, потому что вы не узнаете о возникновении ошибки, однако возвращение лишь части строк приводит к ошибочным результатам. По этой причине важно особенно тщательно тестировать модули-обработчики.

24

Архитектура PL/SQL

Разработчики PL/SQL редко проявляют интерес к архитектуре языка PL/SQL. Типичный представитель нашего сообщества склонен изучить базовый синтаксис PL/SQL, написать свою программу и отправиться домой, чтобы провести побольше времени с семьей или друзьями. И это очень правильный подход!

Тем не менее, на наш взгляд, каждому разработчику желательно хотя бы в общих чертах разбираться в архитектуре PL/SQL. Это позволит не только повысить эффективность использования памяти в программах, но и добиться от программ и приложений того, что на первый взгляд сделать невозможно.

В этой главе вы найдете ответы на следующие вопросы (а также на многие другие):

- Как исполняющее ядро PL/SQL использует память и что можно сделать для управления ее использованием?
- Стоит ли использовать компиляцию в низкоуровневый код или придерживаться используемого по умолчанию режима интерпретации? И вообще — что такое «низкоуровневый код»?
- Почему мои программы вдруг становятся неработоспособными (**INVALID**) и как восстановить их работоспособность?
- Мои таблицы продублированы в 20 разных схемах. Мне действительно придется сопровождать 20 копий своего кода для каждой из этих схем?
- Что такое DIANA?

DIANA

При изучении внутренней структуры компилятора PL/SQL неизменно возникает вопрос: «Что такое DIANA?» Это сокращение (Distributed Intermediate Annotated Notation for Ada, «распределенная промежуточная аннотированная запись для Ada») является частью наследия PL/SQL, произошедшего от языка программирования Ada. У некоторых компиляторов Ada в первой фазе компиляции генерируется код DIANA. Язык PL/SQL тоже изначально генерировал код DIANA в первой фазе процесса компиляции.

Программисту PL/SQL никогда не приходится иметь дело с кодом DIANA. Когда-нибудь компания Oracle, по примеру некоторых разработчиков компиляторов Ada, может решить, что код DIANA стал пережитком прошлого, и использовать другой механизм. Такие изменения во внутренней реализации случаются — например, в сегментном управлении памятью когда-то использовались списки свободных блоков, а теперь используются битовые карты.

Знание DIANA может принести вам репутацию эрудита, но оно не повысит эффективность ваших программ (если вы не занимаетесь разработкой компиляторов PL/SQL).

Как Oracle выполняет код PL/SQL?

Прежде чем переходить к описанию механизма выполнения программ PL/SQL, необходимо определить пару важных понятий:

- **Исполняющее ядро PL/SQL** (виртуальная машина PL/SQL). Так называется компонент Oracle, выполняющий байт-код программ PL/SQL, который при необходимости вызывает сервер базы данных и возвращает результаты вызывающей среде. Исполняющее ядро написано на языке C. Традиционно оно включалось в некоторые клиентские инструменты, например Oracle Forms, где оно открывает сеанс подключения к удаленной базе данных и взаимодействует с ядром SQL через сетевой протокол.
- **Сеанс (Oracle)**. Для большей части (серверного) кода PL/SQL сеансом называется процесс и пространство памяти, связанные с аутентифицированным пользователем. Каждый сеанс имеет собственную область памяти, в которой хранятся данные выполняемой программы. Сеансы начинаются с момента входа пользователя в систему и завершаются при выходе из нее. Для получения информации о текущих сеансах используется представление `V$SESSION`. Oracle поддерживает для сеанса различные виды памяти, включая PGA (Process Global Area), UGA (User Global Area) и CGA (Call Global Area); они будут рассматриваться далее в этом разделе.

Рассмотрим несколько способов выполнения простейшей программы из очень популярной клиентской среды SQL*Plus. Это прекрасный пример приложения, ориентированного на выполнение в рамках сеанса и предоставляющего доступ к окружению PL/SQL, которое входит в состав сервера базы данных. (Об SQL*Plus и его использовании для выполнения кода PL/SQL рассказывается в главе 2.) Конечно, при желании к серверу можно обращаться с вызовами из других приложений (например, из клиентских программ Oracle и даже из таких языков, как Perl, C или Java). Происходящее на сервере практически не зависит от клиентского окружения.

При выполнении кода PL/SQL из SQL*Plus задействован анонимный блок верхнего уровня. Вы, вероятно, знаете, что команда `EXECUTE` в SQL*Plus преобразует вызов программы в анонимный блок, но известно ли вам, что команда `SQL CALL` тоже генерирует упрощенную разновидность анонимного блока? По сути, прежде чем в Oracle9i появилась возможность непосредственного вызова PL/SQL из SQL, все такого рода вызовы производились с использованием анонимных блоков.

Пример

Начнем с простейшего анонимного блока:

```
BEGIN  
  NULL;  
END;
```

Что происходит при его отправке серверу Oracle? Чтобы ответить на этот вопрос, обратимся к рис. 24.1.

Проанализируем процесс, представленный на этом рисунке.

1. Пользователь вводит исходный код блока, а затем отдает SQL*Plus команду на выполнение (косая черта). Как следует из рисунка, SQL*Plus отправляет весь блок, за исключением косой черты, серверу. Передача осуществляется через соединение, установленное для текущего сеанса (например, через Oracle Net или механизм межпроцессных коммуникаций).

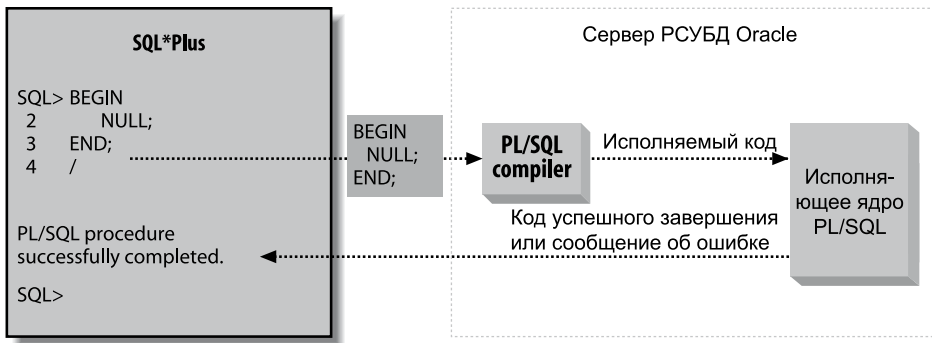


Рис. 24.1. Выполнение тривиального анонимного блока

2. Компилятор PL/SQL пытается откомпилировать полученный анонимный блок в байт-код¹. На первом этапе компилятор проверяет синтаксис и убеждается в том, что программа соответствует грамматике языка. В нашем тривиальном примере код не содержит ни одного идентификатора, а только ключевые слова языка. Если компиляция проходит успешно, Oracle помещает байт-код блока в общую область памяти, а если нет — возвращает сообщения об ошибках сеансу SQL*Plus.
3. Исполняющее ядро PL/SQL интерпретирует байт-код и возвращает сеансу SQL*Plus признак успешного или неудачного завершения сеанса.

Добавим в анонимный блок SQL-запрос и посмотрим, как изменится процесс его выполнения. На рис. 24.2 показаны некоторые компоненты сервера Oracle, участвующие в выполнении команд SQL.

В этом примере значение столбца извлекается из встроенной таблицы DUAL.

Убедившись в том, что код PL/SQL не содержит синтаксических ошибок, компилятор PL/SQL передает код SQL компилятору SQL. Аналогичным образом при вызове программ PL/SQL из команд SQL компилятор SQL передает вызовы PL/SQL компилятору PL/SQL. Компилятор SQL приводит выражения к непосредственно исполняемому виду, анализирует возможности применения кэшированных результатов функций (начиная с Oracle11g), проверяет семантику и синтаксис, проводит разрешение имен и определяет оптимальный план выполнения. Все эти действия являются частью фазы разбора SQL-выражения и предшествуют подстановкам связанных переменных, выполнению и выборке команд SQL.

Хотя PL/SQL использует компилятор SQL совместно с базой данных, это не означает, что в PL/SQL доступны все функции SQL. Например, SQL поддерживает функцию NVL2:

```
SELECT NVL2(NULL, 1, 2) FROM DUAL;
```

Однако попытка вызова NVL2 непосредственно из PL/SQL приводит к ошибке (PLS-00201):

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(NVL2(NULL, 1, 2));
2 BEGIN DBMS_OUTPUT.PUT_LINE (NVL2(NULL, 1, 2)); END;
```

```
*
ERROR at line 1:
ORA-06550: line 1, column 28:
```

продолжение ➤

¹ Если такой же код уже был откомпилирован в текущем сеансе, вполне вероятно, что повторная компиляция не понадобится. Результаты столь затратной операции, как компиляция, сервер кэширует в памяти и старается по возможности использовать повторно.

```
PLS-00201: identifier 'NVL2' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

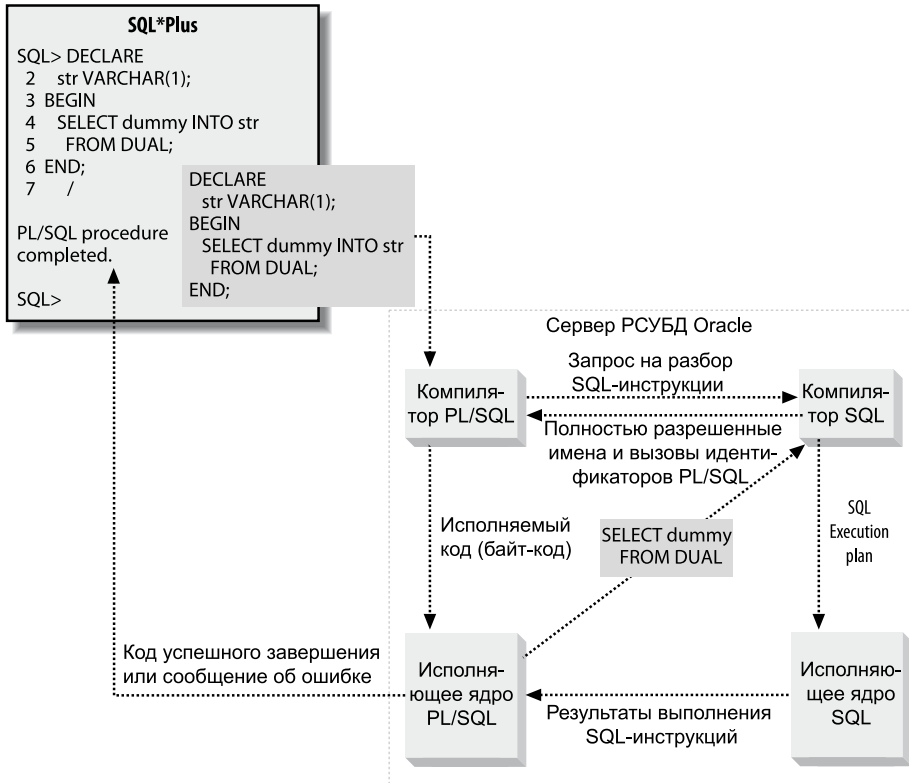


Рис. 24.2. Выполнение анонимного блока, содержащего команду SQL

При компиляции программы PL/SQL встроенные команды SQL подвергаются незначительным преобразованиям: из них удаляются секции INTO, на место локальных переменных программы подставляются связанные значения, удаляются комментарии, а многие идентификаторы (ключевые слова, имена столбцов, имена таблиц и т. д., но не рекомендации и литералы) преобразуются в верхний регистр. Например, если *myvar* является локальной переменной, PL/SQL преобразует команду

```
select dummy into str from dual where dummy = myvar
```

к следующему виду:

```
SELECT DUMMY FROM DUAL WHERE DUMMY = :B1
```

Из кода PL/SQL могут совершаться внешние вызовы двух типов:

- **Хранимые процедуры Java.** Стандартная установка сервера баз данных включает не только виртуальную машину PL/SQL, но и виртуальную машину Java. Вы можете написать спецификацию вызова PL/SQL, логика которого реализуется в виде статического класса Java.
- **Внешние процедуры.** Исполняемую часть подпрограммы PL/SQL также можно реализовать в коде C. Во время работы база данных будет выполнять код в особом процессе и пространстве памяти, отделенных от главного сервера базы данных. Вы отвечаете за поддержку двоичных файлов и наличие копии у каждого узла RAS.

Ограничения компилятора

При компиляции *очень большой* программы PL/SQL от сервера можно получить сообщение об ошибке PLS-00123. Это означает, что компилятор вышел за пределы максимально допустимого количества узлов в дереве разбора. В таком случае программу проще всего разбить на несколько меньших подпрограмм или изменить ее архитектуру (например, использовать временную таблицу вместо 10 000 параметров). Трудно предсказать, сколько узлов понадобится для реализации программы, потому что не каждый узел соответствует какой-либо легко идентифицируемой единице исходного кода (лексеме, строке кода и т. д.).

По данным Oracle, «типичная» хранимая программа создает в среднем четыре узла на каждую строку кода, что дает следующие приблизительные ограничения:

| Тип программы PL/SQL | Приблизительный максимальный размер исходного кода в байтах |
|---|---|
| Тело пакета или типа; автономные функции и процедуры | 256 Мбайт |
| Сигнатура (заголовок) отдельной функции или процедуры | 128 Кбайт |
| Спецификация пакета или типа данных, анонимный блок | 128 Кбайт |

Приведенные значения весьма приблизительны; в реальных программах возможны значительные отклонения в обоих направлениях.

Некоторые из других документированных ограничений компилятора PL/SQL перечислены в следующей таблице.

| Тип программы PL/SQL | Максимальное значение (приблизительное) |
|--|---|
| Уровни вложения блоков | 255 |
| Параметры, передаваемые процедуре или функции | 65 536 |
| Уровни вложения записей | 64 |
| Ссылки на объекты в программах | 65 536 |
| Количество обработчиков исключений в одной программе | 65 536 |
| Точность NUMBER (в знаках) | 38 |
| Размер VARCHAR2 (в байтах) | 32 767 |

Полный список ограничений приведен в приложении к официальной документации *PL/SQL User's Guide and Reference*.

Пакеты по умолчанию

В настоящем объектно-ориентированном языке — таком, как Java, — имеется корневой класс, производными от которого являются все остальные классы. (В Java он называется `Object`.) PL/SQL официально считается объектно-реляционным языком, но по своей сути он является реляционным процедурным языком, а в самой его основе лежит «корневой» пакет с именем `STANDARD`.

Пакеты, которые вы создаете, не являются производными от `STANDARD`, но почти каждая написанная вами программа будет зависеть от этого пакета и использовать его. Собственно, это один из двух используемых по умолчанию пакетов PL/SQL (второй — `DBMS_STANDARD`).

Чтобы лучше понять роль этих пакетов в среде программирования, стоит вернуться к концу 1980-х годов, когда еще не было ни Oracle7, ни SQL*Forms 3, не было даже Oracle PL/SQL. Тогда выяснилось, что SQL — замечательный язык, но его возможности

ограничены. Клиенты Oracle писали программы С, выполнявшие команды SQL, но эти программы приходилось изменять для каждой новой операционной системы.

Было решено создать язык программирования, который мог бы выполнять команды SQL и при этом работать во всех операционных системах, в которых установлена база данных Oracle. Также компания Oracle решила, что вместо изобретения собственного языка она проанализирует существующие языки и определит, не могут ли они стать прототипом для того, чем в будущем стал PL/SQL.

В итоге за прототип был взят язык Ada, изначально разработанный для Министерства обороны США. Конструкция пакетов была позаимствована из Ada. В этом языке можно задать «пакет по умолчанию» для любой программной единицы. Имена элементов пакета по умолчанию не нужно уточнять именем пакета (как в конструкциях вида *мой_пакет.процедура*).

При проектировании PL/SQL идея пакета по умолчанию была сохранена, но способ ее применения несколько изменился. Пользователи PL/SQL не могут задать пакет по умолчанию для программной единицы. В PL/SQL существует два пакета по умолчанию, `STANDARD` и `DBMS_STANDARD`. Они используются по умолчанию во всем языке, а не в конкретной программной единице.

Имена элементов этих пакетов почти всегда используются без уточнения имен. Пакет `STANDARD` объявляет набор типов, исключений и подпрограмм, автоматически доступных в любой программе PL/SQL; многие программисты PL/SQL (ошибочно) считают их «зарезервированными словами» языка. При компиляции кода компилятор Oracle должен разрешить все идентификаторы без уточнения имен; сначала Oracle проверяет, объявлен ли элемент с таким именем в текущей области видимости. Если нет, Oracle проверяет, определен ли элемент с таким именем в пакете `STANDARD` или `DBMS_STANDARD`. Если все идентификаторы будут найдены успешно (и в коде не обнаружены синтаксические ошибки), то код компилируется.

Чтобы понять роль пакета `STANDARD`, рассмотрим следующий, очень странный блок кода PL/SQL. Как вы думаете, что произойдет при выполнении этого блока?

```
/* Файл в Сети: standard_demo.sql */
1 DECLARE
2     SUBTYPE DATE IS NUMBER;
3     VARCHAR2 DATE := 11111;
4     TO_CHAR      PLS_INTEGER;
5     NO_DATA_FOUND EXCEPTION;
6 BEGIN
7     SELECT 1 INTO TO_CHAR
8     FROM SYS.DUAL WHERE 1 = 2;
9 EXCEPTION
10    WHEN NO_DATA_FOUND
11    THEN
12        DBMS_OUTPUT.put_line ('Trapped!');
13 END;
```

Большинство разработчиков PL/SQL скажет: «Этот блок не откомпилируется» или «Будет выведено сообщение «Trapped!», потому что 1 не равно 2».

В действительности блок откомпилируется, но при его выполнении произойдет необработанное исключение `NO_DATA_FOUND`:

```
ORA-01403: no data found
ORA-06512: at line 7
```

Странно, не правда ли? `NO_DATA_FOUND`, единственное исключение, которое обрабатывается в программе, — как оно осталось необработанным? Вопрос в том, какое исключение `NO_DATA_FOUND` обрабатывается в программе? В этом блоке мы объявляем собственное

исключение с именем `NO_DATA_FOUND`. Это имя не является зарезервированным словом языка PL/SQL (в отличие, скажем, от `BEGIN` — объявить переменную с именем `BEGIN` невозможно). В спецификации пакета `STANDARD` одноименное исключение определяется следующим образом:

```
NO_DATA_FOUND exception;
pragma EXCEPTION_INIT(NO_DATA_FOUND, 100);
```

Так как в нашем блоке имеется локально объявленное исключение с именем `NO_DATA_FOUND`, любые *неуточненные* ссылки на этот идентификатор в блоке считаются относящимися именно к нему, а не к исключению пакета `STANDARD`. Если команда `SELECT INTO` не находит ни одной строки, инициируется исключение `STANDARD.NO_DATA_FOUND` — это совсем не то исключение, которое обрабатывается в разделе исключений.

С другой стороны, если привести строку 12 в разделе исключений к следующему виду:

```
WHEN STANDARD.NO_DATA_FOUND
```

то исключение будет успешно обработано, а программа выведет слово «Trapped!».

Кроме неоднозначной ситуации с `NO_DATA_FOUND`, следующие строки тоже выглядят довольно странно.

| Строки | Описание |
|--------|--|
| 2 | Определение нового типа данных «DATE», который на самом деле относится к типу NUMBER |
| 3 | Объявление переменной «VARCHAR2» типа DATA и присваивание ей значения 11111 |
| 4 | Объявление переменной «TO_CHAR» типа PLS_INTEGER |

Мы можем «использовать заново» эти имена «встроенных» элементов языка PL/SQL, потому что все они определяются в пакете `STANDARD`. Эти имена не являются зарезервированными словами PL/SQL; их просто и удобно использовать без указания имени пакета.

Пакет `STANDARD` содержит определения типов данных, поддерживаемых в PL/SQL, заранее определенных исключений и встроенных функций (`TO_CHAR`, `SYSDATE`, `USER` и т. д.). Пакет `DBMS_STANDARD` содержит элементы, относящиеся к работе с транзакциями (такие, как `COMMIT` и `ROLLBACK`), а также функции триггерных событий `INSERTING`, `DELETING` и `UPDATING`.

Несколько замечаний по поводу `STANDARD`:

- Никогда не изменяйте содержимое этого пакета. А если вы это сделаете, не обращайтесь в службу поддержки Oracle за помощью — ведь вы нарушили соглашение о сопровождении продукта! Чтобы вы могли изучить содержимое этого пакета и других пакетов (таких, как `DBMS_OUTPUT` — см. `dbmsotpt.sql`) и `DBMS_UTILITY` (см. `dbmsutil.sql`), администратор базы данных *должен* предоставить вам доступ только для чтения для каталога RDBMS/Admin.
- Oracle даже позволяет прочитать тело пакета `STANDARD`; тела большинства пакетов (например, `DBMS_SQL`) скрываются или защищаются псевдошифрованием. Загляните в сценарий `stdbody.sql`; вы увидите, например, что функция `USER` *всегда* выполняет `SELECT` из `SYS.dual`, а `SYSDATE` выполняет запрос только в том случае, если при выполнении программы `S` для получения системной временной метки происходит ошибка.
- Присутствие некоторой конструкции в `STANDARD` еще не означает, что вы можете использовать точно такой же код в своих блоках PL/SQL. Например, вам не удастся объявить подтип с диапазоном значений, как это делается для `BINARY_INTEGER`.
- Если некоторый элемент определяется в `STANDARD`, это еще не означает, что его можно использовать в PL/SQL. Например, функция `DECODE` объявлена в `STANDARD`, но вызывать ее можно только из команд SQL.

Пакет STANDARD определяется в файлах `stdspec.sql` и `stdbody.sql` из каталога `$ORACLE_HOME/RDBMS/Admin` (в ранних версиях базы данных этот пакет находился в файле `standard.sql`). Пакет `DBMS_STANDARD` определяется в файле `dbmsstdx.sql`.

Если вам интересно, какие из многочисленных предопределенных идентификаторов являются зарезервированными словами в языке PL/SQL, воспользуйтесь сценарием `reserved_words.sql` на сайте книги. Этот сценарий объясняется в главе 3.

Модели разрешений

Oracle поддерживает две модели разрешений доступа. По умолчанию используется *модель прав создателя* (она была единственной до выхода Oracle8i). В этой модели хранящая программа работает с правами своего владельца (или создателя). Другая модель разрешений использует привилегии пользователя, вызвавшего программу; она называется *моделью прав вызывающего*.

Очень важно понимать особенности обеих моделей, потому что во многих приложениях PL/SQL возможно их сочетание. Давайте изучим эти модели более подробно — это поможет вам определить, когда должна использоваться каждая модель.

Модель разрешений создателя

Перед выполнением хранящая программа должна быть откомпилирована и записана в базу данных. Поэтому она всегда сохраняется в конкретной схеме или с конкретной учетной записью пользователя, даже если в ней имеются ссылки на объекты других схем.

При использовании *модели разрешений создателя* действуют следующие правила:

- Разрешение всех внешних ссылок в программе выполняется во время компиляции, с использованием непосредственно предоставленных привилегий схемы, в которой эта программа компилируется.
- Роли базы данных полностью игнорируются при компиляции хранимых программ. Все привилегии, необходимые программе, должны предоставляться непосредственно создателю (владельцу) программы.
- При запуске программы, откомпилированной с использованием модели разрешений создателя (по умолчанию), ее команды SQL выполняются с разрешениями схемы, которой принадлежит эта программа.
- Хотя на выполнение компиляции программы требуется особое разрешение, привилегию `EXECUTE`, позволяющую выполнять эту программу, можно предоставить другим схемам и ролям.

На рис. 24.3 показано, как использовать модель разрешений создателя для управления доступом к объектам базы данных. Все данные заказа хранятся в схеме `OEData`, а код ввода заказов хранится в схеме `OECode`. Владелец схемы `OECode` предоставляются непосредственные привилегии, необходимые для компиляции пакета `Order_Mgt` и позволяющие вводить и отменять заказы.

Для обеспечения корректного обновления таблицы заказов необходимо запретить прямой доступ к ней (через роли или привилегии) из всех схем, кроме `OECode`. Предположим, что из схемы `Sam_Sales` нужно просмотреть все открытые заказы и закрыть те из них, которые были сделаны раньше определенной даты. Ассоциированный с этой схемой пользователь `Sam` не может выполнить команду `DELETE` из процедуры `Close_Old_Orders`; вместо этого он должен вызвать процедуру `Order_Mgt.cancel`.

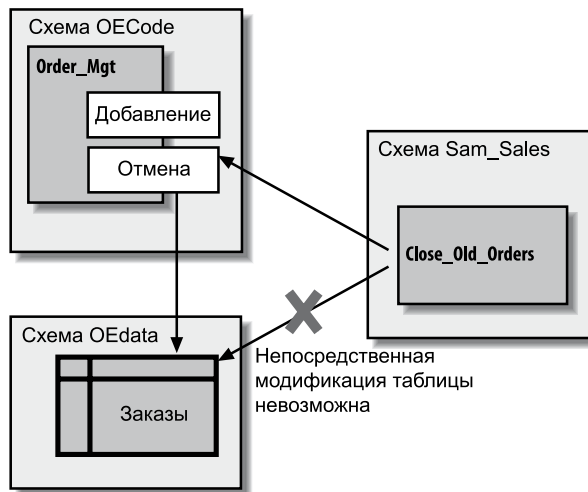


Рис. 24.3. Управление доступом к данным в модели разрешений создателя

Преимущества модели разрешений создателя

В некоторых ситуациях просто не обойтись без модели разрешений создателя, обладающей рядом преимуществ в сравнении с моделью разрешений вызывающего:

- Разработчик имеет больше возможностей для управления доступом к структурам данных и может быть уверен, что изменение содержимого таблицы может осуществляться только через созданный им программный интерфейс (обычно пакет).
- Производительность приложения значительно повышается, поскольку ядру PL/SQL не приходится во время выполнения проверять наличие разрешений на доступ к объектам или, что не менее важно, выяснять, с какой именно таблицей должна работать программа (поскольку одна таблица `accounts` может очень сильно отличаться от другой одноименной таблицы).
- Не нужно беспокоиться о том, что программа обратится не к той таблице. При использовании разрешений создателя программный код будет обрабатывать те же структуры данных, с которыми работали бы команды SQL, выполняемые из SQL*Plus (или другого приложения).

Недостатки модели разрешений создателя

Впрочем, у модели создателя также имеются свои недостатки.

Куда исчезла таблица?

Попытаемся разобраться, какое значение имеют перечисленные правила использования модели разрешений создателя для разработчика программ PL/SQL в его повседневной работе. Разработчики нередко пишут код, который выполняет обработку таблиц и представлений из других схем, создавая для них общие синонимы, которые попросту скрывают схемы. Привилегии при этом предоставляются через роли базы данных.

Эта распространенная модель имеет несколько неприятных особенностей. Предположим, что в сети вашей компании управление доступом к объектам реализовано посредством ролей. При работе с таблицей `accounts` вы можете выполнить в SQL*Plus такой запрос:

```
SQL> SELECT account#, name FROM accounts;
```

Однако если эту же таблицу (и даже тот же запрос) использовать в процедуре, PL/SQL выдаст сразу два сообщения об ошибках:

```
SQL> CREATE OR REPLACE PROCEDURE show_accounts
2  IS
3  BEGIN
4      FOR rec IN (SELECT account#, name FROM accounts)
5      LOOP
6          DBMS_OUTPUT.PUT_LINE (rec.name);
7      END LOOP;
8  END;
9  /
```

Warning: Procedure created with compilation errors.

```
SQL> sho err
Errors for PROCEDURE SHOW_ACCOUNTS:
```

```
LINE/COL ERROR
-----
4/16      PL/SQL: SQL Statement ignored
4/43      PLS-00201: identifier 'ACCOUNTS' must be declared
```

Странно, не правда ли? Проблема заключается в том, что таблица ACCOUNTS на самом деле входит в состав другой схемы; для получения доступа к данным вы использовали синонимы и роли, но это не сработало. Так что если вы когда-нибудь столкнетесь с подобной ситуацией, просто обратитесь к администратору базы данных или владельцу объекта и попросите предоставить вам необходимые привилегии.

Проблемы с сопровождением кода

Предположим, в каждом филиале вашей компании используется собственная схема базы данных. Вы построили большой объем кода, который используется каждым филиалом для сбора и обработки своих, характерных только для него данных. Во всех схемах имеются собственные таблицы с одинаковой структурой, но с разными данными. Такая архитектура была выбрана как для обеспечения безопасности данных, так и для упрощения их передачи через промежуточные табличные пространства.

Теперь нам хотелось бы адаптировать этот код, чтобы сопровождение приложения требовало минимума времени и усилий. Для этого можно поместить его в одну схему и предоставить доступ к нему из объектов всех схем филиалов.

К сожалению, при использовании модели разрешений создателя так поступить невозможно. Если поместить код в определенную схему и задать для объектов всех схем филиалов право EXECUTE, то все они смогут работать с любым набором таблиц, указанных в центральной схеме (с набором одного филиала или, что более вероятно, с набором фиктивных таблиц-заготовок). Это очевидно нежелательно. Придется добавлять весь код в каждую региональную схему, как показано на рис. 24.4.

Не стоит и говорить, каким кошмаром станет сопровождение и модификация такого кода. Конечно, модель разрешений вызывающего обеспечит более удобное решение.

Динамический SQL и разрешения создателя

Еще одним типичным источником проблем, связанных с разрешениями создателя, является динамический SQL (см. главу 16). Предположим, мы создаем обобщенную программу для выполнения команд DDL (да, с точки зрения безопасности эта идея нежелательна, но она демонстрирует нежелательные последствия этого учебного примера):

```
/* Файл в Сети: execddl.sp */
PROCEDURE execDDL (ddl_string IN VARCHAR2)
```

```
IS
BEGIN
    EXECUTE IMMEDIATE ddl_string;
END;
```

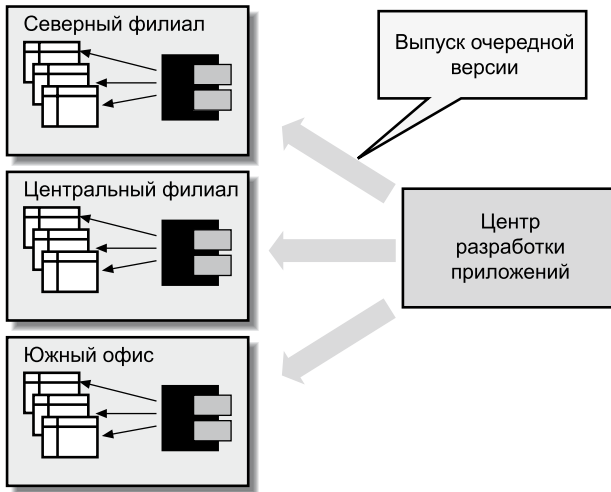


Рис. 24.4. Дублирование кода при использовании модели разрешений создателя

Протестировав этот код в своей схеме, мы решаем, что им могут пользоваться и другие разработчики. Откомпилировав его в схеме **COMMON** (которая используется для всего общедоступного кода), мы передаем разрешение на его выполнение пользователю **PUBLIC**, создаем общий синоним и рассылаем по электронной почте сообщение о новой замечательной программе.

Несколько недель спустя начинают поступать сообщения от коллег: «Я просил создать таблицу, и код выполнялся без ошибок, но таблицы почему-то нет» или «Я просил удалить таблицу, но программа ответила, что такой таблицы не существует, хотя **DESCRIBE** для нее выполняется». В общем, идея ясна. Получив доступ к объектам схемы **COMMON**, вы обнаруживаете там все объекты, которые пользователи пытались создать с помощью его процедуры. Оказывается, если при вызове процедуры **execddl** пользователь попросит создать таблицу и не укажет имя собственной схемы, процедура создаст таблицу в схеме **COMMON**. Иначе говоря, вызов

```
EXEC execddl ('CREATE TABLE newone (rightnow DATE)')
```

создаст таблицу **newone** в схеме **COMMON**. Вызов вида

```
EXEC execddl ('CREATE TABLE scott.newone (rightnow DATE)')
```

мог бы решить проблему, но он завершается сообщением об ошибке (если только для схемы **COMMON** не предоставлена привилегия **CREATE ANY TABLE**). Увы, наша попытка поделиться с другими полезным кодом очень быстро усложнилась. А как было бы здорово, если бы все могли запускать ее с собственными привилегиями (а не с привилегиями схемы **COMMON**), не устанавливая собственную копию кода этой процедуры.

Повышение привилегий и внедрение SQL

Если ваша программа выполняет динамический код **SQL**, зависящий от явно предоставляемых привилегий владельца, а вы полагаете, что модель прав создателя более уместна, — сделайте паузу и обсудите свое решение с коллегами. Старайтесь избегать создания процедур (вроде приведенной выше) в схеме **SYS** или любой другой схеме

с системными привилегиями — они могут послужить «черным ходом» для повышения привилегий при передаче непредвиденного, но формально допустимого ввода.

За дополнительной информацией о борьбе с внедрениями кода обращайтесь к главе 16.

Модель разрешений вызывающего

Иногда программы должны выполняться с привилегиями пользователя, запустившего программу, а не владельца этой программы. В таких случаях следует выбирать модель разрешений вызывающего. Суть ее заключается в том, что разрешение всех внешних ссылок в командах SQL и программах PL/SQL выполняется в соответствии с привилегиями схемы вызывающего программу пользователя, а не владельца схемы, в которой эта программа определена.

Отличие модели разрешений вызывающего от модели разрешений создателя показано на рис. 24.5. Вспомните, что на рис. 24.4 для получения возможности обрабатывать таблицу копии приложения приходилось включить в схемы всех филиалов. С моделью разрешений вызывающего такая необходимость отпадает. Теперь весь код можно разместить в едином централизованном хранилище. И когда пользователь из Северного филиала вызовет централизованно хранящуюся программу (возможно, воспользовавшись синонимом), он автоматически получит доступ к таблицам соответствующей схемы.

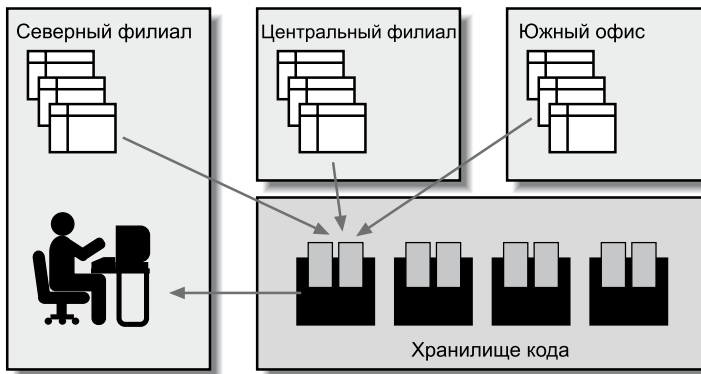


Рис. 24.5. Модель разрешений вызывающего

Как видите, идея модели разрешений вызывающего проста. Давайте посмотрим, как она реализуется программно, а затем разберемся в том, как ее лучше использовать.

Синтаксис модели разрешений вызывающего

Синтаксис поддержки разрешений вызывающего очень прост. В заголовок программы, перед ключевым словом IS или AS, включается следующая конструкция:

```
AUTHID CURRENT_USER
```

В качестве примера рассмотрим все ту же процедуру «exec DDL», но на этот раз объявленную с разрешениями вызывающего:

```
/* Файл в Сети: execddl.sql */
PROCEDURE execddl (ddl_in in VARCHAR2)
  AUTHID CURRENT_USER
IS
BEGIN
  EXECUTE IMMEDIATE ddl_in;
END;
```

Секция `AUTHID CURRENT_USER` перед ключевым словом `IS` указывает, что при выполнении процедуры `execDDL` должны использоваться привилегии вызывающего или текущего пользователя, но не привилегии ее создателя. Если не включить в программу секцию `AUTHID` или задать в ней параметр `AUTHID DEFINER`, разрешение ссылок будет выполняться в соответствии с привилегиями схемы-владельца программы.

РАЗРЕШЕНИЯ ВЫЗЫВАЮЩЕГО ДЛЯ ДИНАМИЧЕСКОГО SQL

Автор написал сотни программ с использованием динамического SQL, и при использовании модели разрешений создателя ему приходилось постоянно беспокоиться о разрешениях, функциональных возможностях и содержимом схем: в какой схеме выполняется программа? кто ее вызывает? что произойдет, если кто-нибудь ее запустит?

Может, у вас появится искушение включить предложение `AUTHID CURRENT_USER` в каждую хранимую программу, в которой используется динамический SQL. Сделав так, вы сможете пребывать в уверенности, что в какой бы схеме ни была откомпилирована программа и кто бы ее ни запустил, она всегда будет работать в текущей схеме.

При этом возникает пара проблем. Во-первых, ваши пользователи теперь должны обладать теми же привилегиями, что и программы (а вы не хотите, чтобы секретарь из отдела кадров мог изменять зарплаты за пределами программы, в которой он работает). Во-вторых, для программ с разрешениями вызывающего база данных должна выполнять дополнительные проверки, что может отрицательно сказаться на производительности системы. Короче говоря, к выбору модели следует подходить сознательно.

Ограничения на использование модели разрешений вызывающего

При использовании модели разрешений вызывающего необходимо помнить о некоторых правилах и ограничениях:

- По умолчанию в программах действует режим `AUTHID DEFINER`.
- В случае применения любых SQL-ссылок на объекты базы данных модель разрешений вызывающего требует проверки во время выполнения программы привилегий, предоставленных вызывающему.
- При использовании модели разрешений вызывающего роли действуют во время выполнения программы, если только программа с разрешениями вызывающего не была вызвана из программы с разрешениями создателя.
- Секция `AUTHID` может содержаться только в заголовках отдельных подпрограмм (процедур или функций), в спецификации пакета или объектного типа. Она не может задаваться в отдельных программах пакета или методах объектного типа. Таким образом, весь пакет работает с правами вызывающего или весь пакет работает с правами создателя. Если одна часть пакета должна выполняться с правами вызывающего, а другая — с правами создателя, значит, придется создать два пакета.
- Разрешение внешних ссылок с привилегиями вызывающего будет действовать для следующих видов команд:
 - `SELECT`, `INSERT`, `UPDATE`, `MERGE` и `DELETE`;
 - команды управления транзакциями `LOCK TABLE`;
 - команд управления курсорами `OPEN` и `OPEN FOR`;
 - команд динамического SQL `EXECUTE IMMEDIATE` и `OPEN FOR USING`;
 - команд SQL, обработанных процедурой `DBMS_SQL.PARSE`.
- Для разрешения всех внешних ссылок на программы PL/SQL и методы объектных типов во время компиляции всегда используются привилегии создателя.

- Привилегии вызывающего можно использовать для изменения разрешения ссылок на статические элементы данных (таблицы и представления).

Привилегии вызывающего могут использоваться для разрешения внешних ссылок на программы PL/SQL. Пример:

```
EXECUTE IMMEDIATE 'BEGIN программа; END;';
```

В этой команде ссылка *программа* будет разрешена во время выполнения в соответствии с привилегиями и пространством имен вызывающего. Также вместо анонимного блока можно использовать команду SQL CALL. (Непосредственно в PL/SQL ее применять нельзя, поскольку она здесь не поддерживается.)

Комбинированная модель разрешений

Как вы думаете, что произойдет, если программа с разрешениями создателя вызовет программу с разрешениями вызывающего? А если наоборот? Правила просты:

- Если программа с разрешениями создателя вызывает программу с разрешениями вызывающего, то при выполнении такой программы будут действовать привилегии вызывающей программы.
- Если программа с разрешениями вызывающего вызывает программу с разрешениями создателя, то при выполнении такой программы будут действовать привилегии ее владельца. Когда управление будет возвращено вызывающей программе, возобновится действие разрешений вызывающего.

Просто запомните, что разрешения создателя «сильнее» разрешений вызывающего.

На сайте книги доступны файлы, при помощи которых можно исследовать нюансы модели прав вызывающего:

- `invdefinv.sql` и `invdefinv.tst` — два сценария, демонстрирующих приоритет прав создателя перед правами вызывающего.
- `invdef_overhead.tst` — анализ дополнительных затрат при использовании прав вызывающего (подсказка: разрешение во время выполнения работает медленнее разрешения во время компиляции).
- `invrole.sql` — демонстрация того, как изменения в ролях могут влиять на разрешение объектных ссылок во время выполнения.
- `irdynsql.sql` — анализ некоторых сложностей, связанных с использованием прав создателя и вызывающего в динамическом SQL.

Назначение ролей программам PL/SQL (Oracle Database 12c)

До выхода Oracle Database 12c программа с правами создателя (определенная с `AUTHID DEFINER` или `AUTHID`) всегда выполнялась с привилегиями стороны, определившей программу. Программа с правами вызывающего (с `AUTHID CURRENT_USER`) всегда выполнялась с привилегиями стороны, вызвавшей программу.

Из существования двух разных режимов `AUTHID` следовало, что любая программа, предназначенная для выполнения всеми пользователями, должна была определяться с правами создателя. Затем она выполнялась со всеми привилегиями создателя, что могло оказаться неоптимальным с точки зрения безопасности.

В Oracle Database 12c появилась возможность назначения ролей пакетам PL/SQL и процедурам/функциям уровня схемы. Привилегии программ на базе ролей позволяют разработчику оптимизировать привилегии, доступные для стороны, вызвавшей программу.

Теперь программу можно определить с правами вызывающего, а затем дополнить привилегии вызывающего более конкретными и ограниченными привилегиями, предоставляемыми ролью.

Следующий пример показывает, как назначать роли программам и как они влияют на выполнение программ. Предположим, схема HR содержит таблицы `employees` и `departments`, которые определяются и заполняются данными:

```
CREATE TABLE departments(
  department_id    INTEGER,
  department_name  VARCHAR2 (100),
  staff_freeze     CHAR (1)
)
/
BEGIN
  INSERT INTO departments
    VALUES (10, 'IT', 'Y');
  INSERT INTO departments
    VALUES (20, 'HR', 'N');
  COMMIT;
END;
/
CREATE TABLE employees
(
  employee_id      INTEGER,
  department_id    INTEGER,
  last_name        VARCHAR2 (100)
)
/
BEGIN
  DELETE FROM employees;
  INSERT INTO employees VALUES (100, 10, 'Price');
  INSERT INTO employees VALUES (101, 20, 'Sam');
  INSERT INTO employees VALUES (102, 20, 'Joseph');
  INSERT INTO employees VALUES (103, 20, 'Smith');
  COMMIT;
END;
/
```

Схема SCOTT содержит *только* таблицу `employees`:

```
CREATE TABLE employees
(
  employee_id      INTEGER,
  department_id    INTEGER,
  last_name        VARCHAR2 (100)
)
/
BEGIN
  DELETE FROM employees;
  INSERT INTO employees VALUES (100, 10, 'Price');
  INSERT INTO employees VALUES (101, 20, 'Sam');
  INSERT INTO employees VALUES (102, 20, 'Joseph');
  INSERT INTO employees VALUES (103, 20, 'Smith');
  COMMIT;
END;
/
```

Схема HR также содержит процедуру для удаления всех работников заданного отдела при условии, что штат отдела не «заморожен». Сначала я определяю эту процедуру с правами создателя:

```
CREATE OR REPLACE PROCEDURE remove_emps_in_dept (
  department_id_in IN employees.department_id%TYPE)
  AUTHID DEFINER
IS
  l_freeze    departments.staff_freeze%TYPE;
BEGIN
  SELECT staff_freeze
    INTO l_freeze
```

```

        FROM HR.departments
        WHERE department_id = department_id_in;
    IF l_freeze = 'N'
    THEN
        DELETE FROM employees
            WHERE department_id =
                department_id_in;
    END IF;
END;
/

```

Схеме SCOTT разрешается выполнение этой процедуры:

```

GRANT EXECUTE
    ON remove_emps_in_dept
    TO SCOTT
/

```

Когда SCOTT выполняет процедуру так, как показано ниже, удаляются три строки — но из таблицы `employees` схемы HR, поскольку процедура объявлена с правами создателя:

```

BEGIN
    HR.remove_emps_in_dept (20);
END;
/

```

Процедуру нужно изменить так, чтобы строки удалялись из таблицы `employees` схемы SCOTT, а не HR. Именно это и делает модель прав создателя. Но если привести секцию AUTHID процедуры к виду:

```
AUTHID CURRENT_USER
```

и снова выполнить процедуру, будет получен следующий результат:

```

BEGIN * ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "HR.REMOVE_EMPS_IN_DEPT", line 7
ORA-06512: at line 2

```

Проблема в том, что Oracle теперь использует привилегии SCOTT для разрешения ссылок на две таблицы: HR.departments и HR.employees. Однако SCOTT не имеет привилегий для таблицы departments схемы HR, поэтому Oracle выдает ошибку ORA-00942. До выхода Oracle Database 12c администратор базы данных должен был предоставить SCOTT необходимые привилегии HR.departments. Теперь вместо этого можно выполнить следующие действия:

1. В схеме с необходимыми привилегиями создайте роль и предоставьте ее HR:

```

CREATE ROLE hr_departments
/
GRANT hr_departments TO hr
/

```

2. Подключитесь к HR, предоставьте нужные привилегии роли и назначьте ее процедуре:

```

GRANT SELECT
    ON departments
    TO hr_departments
/
GRANT hr_departments TO PROCEDURE remove_emps_in_dept
/

```

Теперь при выполнении следующего фрагмента из схемы SCOTT строки будут удаляться правильно:

```

SELECT COUNT (*)
    FROM employees
    WHERE department_id = 20
/

```

```

COUNT(*)
-----
      3

BEGIN
  hr.remove_emps_in_dept (20);
END;
/

SELECT COUNT (*)
  FROM employees
 WHERE department_id = 20
/

COUNT(*)
-----
      0

```

Роли, назначаемые программам, не влияют на компиляцию. Они влияют на проверку привилегий команд SQL, выдаваемых программой во время выполнения. Процедура или функция выполняется с привилегиями обеих своих ролей, а также других активных ролей. Эта возможность в основном используется для программ с правами вызывающего, как в приведенном примере. Скорее всего, для программ с правами создателя возможность назначения ролей будет рассматриваться при выполнении динамического SQL, поскольку привилегии таких динамических команд проверяются во время выполнения.

Функции «Кто меня вызвал?» (Oracle Database 12c)

В Oracle Database 12c появились две новые функции (которые могут вызываться только из команд SQL), которые возвращают информацию о вызывающем пользователе в зависимости от того, используется ли модель прав вызывающего или создателя:

- **ORA_INVOKING_USER** — возвращает имя пользователя, активизирующего текущую команду или представление. Все промежуточные представления рассматриваются в соответствии с их секциями **BEQUEATH**. Если вызывающий пользователь является пользователем, определяемым Oracle Database Real Application Security (еще одна новая возможность 12.1, следующее поколение функциональности виртуальных частных баз данных), то функция возвращает **XS\$NULL**.
- **ORA_INVOKING_USERID** — возвращает идентификатор пользователя, активизирующего текущую команду или представление. Все промежуточные представления рассматриваются в соответствии с их секциями **BEQUEATH**. Если вызывающий пользователь является пользователем, определяемым Oracle Database Real Application Security (еще одна новая возможность 12.1, следующее поколение функциональности виртуальных частных баз данных), то функция возвращает идентификатор, общий для всех сеансов Real Application Security, но отличный от идентификатора любого пользователя базы данных.

Примеры использования этих функций продемонстрированы в следующем разделе, при описании секции **BEQUEATH CURRENT_USER** для представлений.

BEQUEATH CURRENT_USER для представлений (Oracle Database 12c)

До выхода версии Oracle Database 12c функции, выполняемые представлениями, всегда выполнялись с привилегиями владельца представления, а не владельца функции. Таким образом, если функция определялась с моделью прав вызывающего, ее поведение могло сильно отличаться от ожидаемого.

В версии 12c добавилась секция `BEQUEATH` для представлений, которая позволяет определить представление, адаптирующееся к задействованным в нем функциям с правами вызывающего.

Рассмотрим пример использования этой возможности (весь код, приведенный в этом разделе, находится в файле `12c_bequeath.sql` на сайте книги). Я создаю таблицу и функцию в схеме `HR`:

```
CREATE TABLE c12_ems
(
  employee_id    INTEGER,
  department_id  INTEGER,
  last_name      VARCHAR2 (100)
)
/

BEGIN
  INSERT INTO c12_ems VALUES (1, 100, 'abc');
  INSERT INTO c12_ems VALUES (2, 100, 'def');
  INSERT INTO c12_ems VALUES (3, 200, '123');
  COMMIT;
END;
/

CREATE OR REPLACE FUNCTION emps_count (
  department_id_in IN INTEGER)
  RETURN PLS_INTEGER
  AUTHID CURRENT_USER
IS
  l_count    PLS_INTEGER;
  l_user     VARCHAR2 (100);
  l_userid   VARCHAR2 (100);
BEGIN
  SELECT COUNT (*)
    INTO l_count
    FROM c12_ems
   WHERE department_id = department_id_in;

  /* Вывести информацию о вызывающем пользователе */
  SELECT ora_invoking_user, ora_invoking_userid
    INTO l_user, l_userid FROM DUAL;

  DBMS_OUTPUT.put_line ('Invoking user=' || l_user);
  DBMS_OUTPUT.put_line ('Invoking userID=' || l_userid);

  RETURN l_count;
END;
/

Обратите внимание на вызовы двух новых функций: ORA_INVOKING_USER и ORA_INVOKING_USER_ID.

Затем я создаю представление, указывая права вызывающего в BEQUEATH, и обеспечиваю возможность обращения с запросом к этому представлению для SCOTT:
```

```
CREATE OR REPLACE VIEW emp_counts_v
  BEQUEATH CURRENT_USER
AS
  SELECT department_id, emps_count (department_id) emps_in_dept
    FROM c12_ems
  /

GRANT SELECT ON emp_counts_v TO scott
/
```

В схеме SCOTT создается еще одна таблица c12_ems, которая заполняется другими данными:

```
CREATE TABLE c12_ems
(
  employee_id    INTEGER,
  department_id  INTEGER,
  last_name      VARCHAR2 (100)
)
/

BEGIN
  INSERT INTO c12_ems VALUES (1, 200, 'SCOTT.ABC');
  INSERT INTO c12_ems VALUES (2, 200, 'SCOTT.DEF');
  INSERT INTO c12_ems VALUES (3, 400, 'SCOTT.123');
  COMMIT;
END;
/
```

Наконец, я выполняю выборку всех строк представления. Результат запроса:

```
SQL> SELECT * FROM hr.emp_counts_v
2 /
```

| DEPARTMENT_ID | EMPS_IN_DEPT |
|---------------|--------------|
| 100 | 0 |
| 100 | 0 |
| 200 | 2 |

```
SCOTT
107
SCOTT
107
SCOTT
107
```

Как видите, данные, возвращенные представлением, взяты из таблицы HR (идентификатор отдела 100), но сводная информация, возвращаемая вызовом функции, отражает содержимое таблицы SCOTT, а функции ORA_INVOKING* возвращают информацию SCOTT.

Учтите, что конструкция BEQUEATH CURRENT_USER *не превращает* само представление в объект с правами вызывающего. Разрешение имен в представлении по-прежнему осуществляется с использованием схемы владельца представления, а привилегии проверяются с использованием привилегий владельца представления.

Главное преимущество этого синтаксиса заключается в том, что он позволяет таким функциям, как SYS_CONTEXT и USERENV, возвращать логически целостные результаты при вызове из представлений.

Ограничение привилегий в модели прав вызывающего (Oracle Database 12c)

Когда пользователь запускает процедуру или функцию, определенную в программной единице PL/SQL, которая была создана с секцией AUTHID CURRENT_USER, эта подпрограмма временно наследует все привилегии вызывающего пользователя во время выполнения процедуры. Соответственно, в это время *владелец* программы с правами вызывающего получает доступ к привилегиям вызывающего пользователя.

Если привилегии вызывающего превосходят привилегии владельца, возникает риск того, что владелец подпрограммы сможет злоупотребить повышенными привилегиями

вызывающего. Риски такого рода повышаются, когда пользователи приложений получают доступ к базе данных, использующей процедуры с правами вызывающего.

В Oracle Database 12c добавлены две новые привилегии, которые могут управлять доступностью привилегий вызывающего для процедуры владельца: `INHERIT PRIVILEGES` и `INHERIT ANY PRIVILEGES`.

Любой пользователь может предоставить или отозвать привилегию `INHERIT PRIVILEGES` для пользователей, процедуры которых с правами вызывающего он хочет запускать. Привилегия `INHERIT ANY PRIVILEGES` находится под управлением пользователей `SYS`.

Когда пользователь запускает процедуру с правами вызывающего, Oracle проверяет, имеется ли у владельца процедуры привилегия `INHERIT PRIVILEGES` для вызывающего пользователя или же владельцу была предоставлена привилегия `INHERIT ANY PRIVILEGES`. Если проверка привилегий не проходит, Oracle возвращает ошибку `ORA-06598` (недостаточный уровень `INHERIT PRIVILEGES`).

Преимущество этих двух привилегий заключается в том, что они позволяют вызывающему пользователю контролировать доступ к их привилегиям при запуске процедур с правами вызывающего или запросов к представлениям `BEQUEATH CURRENT_USER`.

Всем пользователям по умолчанию предоставляется привилегия `INHERIT PRIVILEGES ON USER` *новый_пользователь* `TO PUBLIC` при создании их учетных записей (или обновлении ранее созданных учетных записей до текущей версии). Это означает, что если в вашей системе не существует риска злоупотребления привилегиями «сильных вызывающих» со стороны «слабых владельцев», все будет работать точно так же, как работало в более ранних версиях Oracle Database.

Вызывающий пользователь может лишить других пользователей привилегии `INHERIT PRIVILEGES` и предоставить ее только доверенным пользователям. Синтаксис предоставления привилегии `INHERIT PRIVILEGES` выглядит так:

```
GRANT INHERIT PRIVILEGES ON USER вызывающий_пользователь TO владелец_процедуры
```

Условная компиляция

Механизм условной компиляции, появившийся в Oracle Database 10g Release 2, позволяет организовать избирательную компиляцию частей программы в зависимости от условий, заданных в директиве `$IF`.

Например, условная компиляция будет очень полезна, если вам потребуется:

- Написать программу, которая работает в разных версиях Oracle и использует специфические возможности этих версий. А конкретнее — если вы хотите использовать новые возможности Oracle там, где они доступны, но при этом ваша программа также должна компилироваться и работать в старых версиях. Без условной компиляции вам придется вести несколько разных файлов или использовать сложную логику подстановки переменных в `SQL*Plus`.
- Выполнять некоторый код во время тестирования и отладки, но пропускать его в окончательной версии. До появления условной компиляции вам пришлось бы либо закрывать нежелательные строки комментариями, либо добавлять лишние проверки в логику приложения — даже в окончательную версию.
- Устанавливать/компилировать разные элементы приложения в зависимости от требований (например, набора компонентов, на которые у пользователя имеется лицензия). Условная компиляция очень сильно упрощает ведение сложной кодовой базы.

Условная компиляция реализуется посредством включения директив компилятора в исходный код. При компиляции программы препроцессор PL/SQL обрабатывает

директивы и выбирает те части кода, которые требуется откомпилировать. «Урезанный» исходный код передается компилятору для дальнейшей обработки.

Директивы делятся на три типа:

- **Директивы выбора** — директива `$IF` проверяет выражение и определяет, какой код следует включить в обработку (или исключить из нее).
- **Директивы получения информации** — синтаксис `$$identifier` используется для обращения к флагам условной компиляции. К директивам получения информации можно обращаться в директиве `$IF` или использовать их независимо в коде.
- **Директивы ошибок** — директива `$ERROR` используется для вывода ошибок компиляции на основании условий, вычисленных при подготовке кода препроцессором.

Сначала я приведу несколько примеров, а затем мы более подробно рассмотрим каждую директиву. Вы также научитесь пользоваться двумя пакетами, связанными с условной компиляцией: `DBMS_DB_VERSION` и `DBMS_PREPROCESSOR`.

Примеры условной компиляции

Начнем с примеров использования разных видов условной компиляции.

Использование констант пакета приложения в директиве `$IF`

Директива `$IF` может обращаться к константам, определенным в ваших пакетах. В следующем примере начисляемая премия изменяется в зависимости от того, соответствует ли место установки стороннего приложения положению закона Сарбейнза-Оксли. Скорее всего, эта настройка будет оставаться неизменной в течение долгого времени. Если я положусь на традиционную условную команду, в приложении останется ветвь логики, которая никогда не будет применяться. С условной компиляцией лишний код удаляется перед компиляцией:

```
/* Файл в Сети: cc_my_package.sql */
PROCEDURE apply_bonus (
    id_in IN employee.employee_id%TYPE
    ,bonus_in IN employee.bonus%TYPE)
IS
BEGIN
    UPDATE employee
    SET bonus =
        $IF employee_rp.apply_sarbanes_oxley
        $THEN
            LEAST (bonus_in, 10000)
        $ELSE
            bonus_in
        $END
    WHERE employee_id = id_in;
    NULL;
END apply_bonus;
```

Переключение состояния трассировки с использованием флагов условной компиляции

Теперь я могу создать собственные механизмы отладки/трассировки и выполнить их условную компиляцию в своем коде. Это означает, что в окончательной версии кода этот код будет полностью удален, чтобы избежать непроизводительных затрат на выполнение этой логики. Следует отметить, что специальный параметр компилятора `PLSQL_CCFLAGS` может задавать как логические значения, так и значения `PLS_INTEGER`:

```
/* Файл в Сети: cc_debug_trace.sql */
ALTER SESSION SET PLSQL_CCFLAGS = 'oe_debug:true, oe_trace_level:10';
```

продолжение ➤

```

PROCEDURE calculate_totals
IS
BEGIN
  IF $$oe_debug AND $$oe_trace_level >= 5
  THEN
    DBMS_OUTPUT.PUT_LINE ('Tracing at level 5 or higher');
  END
  NULL;
END calculate_totals;

```

Директива получения информации

Директива получения информации запрашивает информацию о среде компиляции. Конечно, по такому описанию трудно понять что-то определенное, поэтому мы должны поближе познакомиться с синтаксисом директив получения информации и разными источниками информации для таких директив.

Синтаксис директивы получения информации выглядит так:

\$\$идентификатор — здесь *идентификатор* представляет собой любой идентификатор PL/SQL, который может представлять:

- **Параметры среды компиляции** — значения из представления словаря данных USER_PLSQL_OBJECT_SETTINGS.
- **Пользовательская директива** — определяется командой ALTER...SET PLSQL_CCFLAGS (см. далее «Использование параметра PLSQL_CCFLAGS»).
- **Предопределенные директивы** — \$\$PLSQL_LINE и \$\$PLSQL_UNIT; возвращают номер строки и имя программы.

Директивы получения информации предназначены для использования в конструкциях условной компиляции, но они также могут использоваться в других местах кода PL/SQL. Например, для вывода текущего номера строки в моей программе я могу использовать следующий код:

```
DBMS_OUTPUT.PUT_LINE ($$PLSQL_LINE);
```

Также директивы получения информации могут использоваться для определения и применения констант уровня приложения. Предположим, максимальный период хранения данных в моем приложении составляет 100 лет. Вместо того чтобы жестко фиксировать это значение в своем коде, я делаю следующее:

```

ALTER SESSION SET PLSQL_CCFLAGS = 'max_years:100';
PROCEDURE work_with_data (num_years_in IN PLS_INTEGER)
IS
BEGIN
  IF num_years_in > $$max_years THEN ...
END work_with_data;

```

Что еще важнее, директивы получения информации могут использоваться в тех местах, где переменные запрещены. Пара примеров:

```

DECLARE
  l_big_string VARCHAR2($$MAX_VARCHAR2_SIZE);
  l_default_app_err EXCEPTION;
  PRAGMA EXCEPTION_INIT (l_default_app_err, $$DEF_APP_ERR_CODE);
BEGIN
  The DBMS_DB_VERSION package

```

Встроенный пакет DBMS_DB_VERSION предоставляет набор констант для получения абсолютной и относительной информации о версии установленной базы данных. Константы, определенные в версии 12.1, приведены в табл. 24.1. В каждой новой версии Oracle добавляются две новые относительные константы, а значения, возвращаемые константами VERSION и RELEASE, обновляются.

Таблица 24.1. Константы DBMS_DB_VERSION

| Имя пакетной константы | Описание | Значение в Oracle Database 12c Release 1 |
|-----------------------------|---|--|
| DBMS_DB_VERSION.VERSION | Номер версии базы данных | 12 |
| DBMS_DB_VERSION.RELEASE | Номер выпуска базы данных | 1 |
| DBMS_DB_VERSION.VER_LE_9 | TRUE, если текущая версия меньше либо равна Oracle9i | FALSE |
| DBMS_DB_VERSION.VER_LE_9_1 | TRUE, если текущая версия меньше либо равна 9.1 | FALSE |
| DBMS_DB_VERSION.VER_LE_9_2 | TRUE, если текущая версия меньше либо равна 9.2 | FALSE |
| DBMS_DB_VERSION.VER_LE_10 | TRUE, если текущая версия меньше либо равна Oracle10g | FALSE |
| DBMS_DB_VERSION.VER_LE_10_1 | TRUE, если текущая версия меньше либо равна 10.1 | FALSE |
| DBMS_DB_VERSION.VER_LE_10_2 | TRUE, если текущая версия меньше либо равна 10.2 | FALSE |
| DBMS_DB_VERSION.VER_LE_11_1 | TRUE, если текущая версия меньше либо равна 11.1 | FALSE |
| DBMS_DB_VERSION.VER_LE_11_2 | TRUE, если текущая версия меньше либо равна 11.2 | FALSE |
| DBMS_DB_VERSION.VER_LE_12 | TRUE, если текущая версия меньше либо равна 12.1 | TRUE |
| DBMS_DB_VERSION.VER_LE_12_1 | TRUE, если текущая версия меньше либо равна 12.1 | TRUE |

Этот пакет проектировался для использования с условной компиляцией, но разумеется, ничто не мешает вам использовать его для любых других целей.

Интересно, что вы можете писать выражения со ссылками на еще не определенные константы пакета DBMS_DB_VERSION. Если они не будут обрабатываться, как в следующем примере, это не приведет к ошибкам. Пример:

```

$IF DBMS_DB_VERSION.VER_LE_12_1
$THEN
    Используется этот код.
$ELIF DBMS_DB_VERSION.VER_LE_13
    Заглушка на будущее.
$END

```

Определение параметров среды компиляции

Следующая информация (соответствующая значениям из представления словаря данных USER_PLSQL_OBJECT_SETTINGS) доступна в директивах получения информации:

- \$\$PLSQL_DEBUG — режим отладки для единицы компиляции.
- \$\$PLSQL_OPTIMIZE_LEVEL — уровень оптимизации для единицы компиляции.
- \$\$PLSQL_CODE_TYPE — режим компиляции для единицы компиляции.
- \$\$PLSQL_WARNINGS — режим предупреждений компиляции для единицы компиляции.
- \$\$NLS_LENGTH_SEMANTICS — значение, заданное для семантики длины NLS.

Примеры использования всех перечисленных параметров представлены в файле `ss_plsql_parameters.sql` на сайте книги.

Имя и номер строки единицы компиляции

Oracle неявно определяет четыре очень полезные директивы получения информации, предназначенные для использования в директивах \$IF и \$ERROR:

- `$$PLSQL_UNIT` — имя единицы компиляции, в которой находится ссылка. Если этой единицей является анонимный блок, то `$$PLSQL_UNIT` содержит NULL.
- `$$PLSQL_LINE` — номер строки в единице компиляции, в которой находится ссылка.
- `$$PLSQL_UNIT_OWNER` (Oracle Database 12c) — имя владельца текущей программной единицы PL/SQL. Если этой единицей является анонимный блок, то `$$PLSQL_UNIT_OWNER` содержит NULL.
- `$$PLSQL_UNIT_TYPE` (Oracle Database 12c) — тип текущей программной единицы PL/SQL: `ANONYMOUS BLOCK`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, `TRIGGER`, `TYPE` или `TYPE BODY`. В анонимных блоках или триггерах, не являющихся триггерами DML, `$$PLSQL_UNIT_TYPE` имеет значение `ANONYMOUS BLOCK`.

Для получения текущих номеров строк можно использовать встроенные функции `DBMS_UTILITY.FORMAT_CALL_STACK` и `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`, но тогда вам придется разбирать эти строки для получения имени строки и имени программной единицы. Директивы получения информации позволяют проще получить нужные сведения. Пример:

```
BEGIN
  IF l_balance < 10000
  THEN
    raise_error (
      err_name => 'BALANCE TOO LOW'
      ,failed_in => $$plsql_unit
      ,failed_on => $$plsql_line
    );
  END IF;
  ...
END;
```

Использование последних двух директив продемонстрировано в файле `cc_line_unit.sql` на сайте книги.

Обратите внимание: если директива `$$PLSQL_UNIT` находится внутри пакета, то она возвращает имя пакета, а не имя отдельной процедуры или функции в пакете.

Использование параметра `PLSQL_CCFLAGS`

Oracle также предоставляет параметр `PLSQL_CCFLAGS`, который может использоваться с условной компиляцией. По сути он позволяет определять пары «имя/значение», причем имя в дальнейшем может использоваться в директивах получения информации в логике условной компиляции. Пример:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'use_debug:TRUE, trace_level:10';
```

Имя флага представляет собой любой допустимый идентификатор PL/SQL, включая зарезервированные и ключевые слова (перед идентификатором ставится префикс `$$`, исключающий путаницу с обычным кодом PL/SQL). Имени может присваиваться один из следующих литералов: `TRUE`, `FALSE`, `NULL` или `PLS_INTEGER`.

Значение `PLSQL_CCFLAGS` будет ассоциироваться с каждой программой, компилируемой в этом сеансе. Если вы хотите сохранить свои настройки в программе, то будущие компиляции командой `ALTER...COMPILE` должны включать секцию `REUSE SETTINGS`.

Так как вы можете изменить значение параметра, а затем откомпилировать выбранные программные единицы, это позволяет легко определять разные наборы директив получения информации для разных программ.

Обратите внимание: директива позволяет ссылаться на флаги, не определенные в `PLSQL_CCFLAGS`; результат будет равен NULL. Если включить предупреждения компилятора, при ссылке на неопределенный флаг база данных вернет предупреждение PLW-06003.

Директива \$IF

Конструкция выбора, реализованная в виде директивы **\$IF**, используется для передачи шага условной компиляции препроцессору. Общий синтаксис этой директивы:

```
$IF логическое_выражение
$THEN
    фрагмент_кода
[$ELIF логическое_выражение
$THEN
    фрагмент_кода]
[$ELSE
    фрагмент_кода]
$END
```

Здесь *логическое_выражение* представляет собой статическое выражение (которое может вычисляться на момент компиляции), результатом которого является TRUE, FALSE или NULL. *Фрагмент_кода* содержит произвольную последовательность команд PL/SQL, которая передается компилятору для обработки (в соответствии с правилами вычисления выражений).

Статические выражения могут строиться из следующих элементов:

- Логические значения, PLS_INTEGER и NULL, а также комбинации этих литералов.
- Статические выражения с логическими значениями, PLS_INTEGER и VARCHAR2.
- Директивы получения информации (то есть идентификаторы с префиксом \$\$). Такие директивы могут предоставляться Oracle (например, \$\$PLSQL_OPTIMIZE_LEVEL; полный список приведен в разделе «Оптимизирующий компилятор» главы 21) или задаваться параметром компиляции PLSQL_CCFLAGS (см. ранее в этой главе).
- Статические константы, определенные в пакете PL/SQL.
- Большинство операторов сравнения (>, <, = и <> допустимы, но использовать выражение IN нельзя), логические операции — такие, как AND и OR, конкатенации статических символьных выражений и проверки на NULL.

Статическое выражение не может содержать вызовы процедур или функций, требующих выполнения; они не могут вычисляться во время компиляции, а следовательно, сделают недействительным выражение в директиве **\$IF**. Компилятор в таком случае выдает сообщение об ошибке (PLS-00174).

Несколько примеров статических выражений в директивах **\$IF**:

- Если пользовательская директива получения информации, управляющая отладкой, отлична от NULL, инициализировать подсистему отладки:

```
$IF $$app_debug_level IS NOT NULL $THEN
    debug_pkg.initialize;
$END
```

- Проверка значения пользовательской пакетной константы и уровня оптимизации:

```
$IF $$PLSQL_OPTIMIZE_LEVEL = 2 AND appdef_pkg.long_compilation
$THEN
    $ERROR 'Do not use optimization level 2 for this program!' $END
$END
```



Строковые литералы и конкатенация строк разрешены только в директиве **\$ERROR**; в директиве **\$IF** их присутствие недопустимо.

Директива \$ERROR

Директива \$ERROR инициирует сбой в текущей единице компиляции и возвращает указанное сообщение об ошибке. Синтаксис директивы:

```
$ERROR выражение_VARCHAR2 $END
```

Предположим, я хочу задать для некоторой программной единицы уровень оптимизации 1, чтобы ускорить компиляцию. В следующем примере директива \$\$ используется для проверки уровня оптимизации, предоставляемого средой компиляции. Далее при необходимости программа инициирует ошибку директивой \$ERROR:

```
/* Файл в Сети: cc_opt_level_check.sql */
SQL> CREATE OR REPLACE PROCEDURE long_compilation
  2  IS
  3  BEGIN
  4  $IF $$plsql_optimize_level != 1
  5  $THEN
  6    $ERROR 'This program must be compiled with optimization level = 1' $END
  7  $END
  8    NULL;
  9  END long_compilation;
 10  /
```

Warning: Procedure created with compilation errors.

```
SQL> SHOW ERRORS
Errors for PROCEDURE LONG_COMPILATION:
```

```
LINE/COL ERROR
-----
6/4      PLS-00179: $ERROR: This program must be compiled with
         optimization level = 1
```

Синхронизация кода с использованием пакетных констант

Использование пакетных констант в директиве выбора позволяет легко синхронизировать несколько программных единиц по некоторому параметру условной компиляции. Такая синхронизация возможна благодаря тому, что автоматическое управление зависимостями Oracle распространяется на директивы выбора. Иначе говоря, если программная единица PROG содержит директиву выбора, которая ссылается на пакет PKG, то она помечается как зависящая от PKG. Когда спецификация PKG перекомпилируется, все программные единицы, использующие пакетную константу, помечаются как INVALID и должны быть перекомпилированы.

Допустим, я хочу использовать условную компиляцию для автоматического включения или исключения логики отладки и трассировки в своей кодовой базе. Я определяю спецификацию пакета для хранения необходимых констант:

```
/* Файл в Сети: cc_debug.pks */
PACKAGE cc_debug
IS
  debug_active CONSTANT BOOLEAN := TRUE;
  trace_level  CONSTANT PLS_INTEGER := 10;
END cc_debug;
```

Затем эти константы используются в процедуре calc_totals:

```
PROCEDURE calc_totals
IS
BEGIN
$IF cc_debug.debug_active AND cc_debug.trace_level > 5 $THEN
```

```

    log_info (...);
$END
...
END calc_totals;

```

В процессе разработки константа `debug_active` инициализируется значением `TRUE`. Когда нужно переводить код в окончательную версию, я меняю значение флага на `FALSE` и перекомпилирую пакет. Программа `calc_totals` и все остальные программы с похожими директивами выбора объявляются неработоспособными и должны пройти перекомпиляцию.

Применение директив получения информации для определения конфигурации конкретных программ

Пакетные константы удобны для координации настроек между несколькими программными единицами. Директивы получения информации лучше подходят в ситуациях, когда требуется применить разные настройки для разных программ.

После того как программа будет откомпилирована с некоторым набором значений, она сохраняет эти значения до следующей компиляции (из файла или простой перекомпиляции командой `ALTER...COMPILE`). Кроме того, программа будет заведомо перекомпилирована с тем же исходным кодом, который был выбран при *предыдущей* компиляции, если истинны *все* следующие условия:

- Ни одна из директив условной компиляции не ссылается на пакетные константы (а только на директивы получения информации).
- При перекомпиляции программы используется секция `REUSE SETTINGS`, а параметр `PLSQL_CCFLAGS` не включен в команду `ALTER...COMPILE`.

Данная возможность продемонстрирована в сценарии `cc_reuse_settings.sql` (на сайте книги); результат выполнения этого сценария приведен ниже. Сначала я присваиваю `app_debug` значение `TRUE`, а затем перекомпилирую программу с этим значением. Запрос к `USER_PLSQL_OBJECT_SETTINGS` показывает, что это значение теперь связано с программной единицей:

```
/* Файл в Сети: cc_reuse_settings.sql */
```

```

SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:TRUE';
SQL> CREATE OR REPLACE PROCEDURE test_ccflags
2  IS
3  BEGIN
4      NULL;
5  END test_ccflags;
6  /

```

```

SQL> SELECT name, plsql_ccflags
2  FROM user_plsql_object_settings
3  WHERE NAME LIKE '%CCFLAGS%';

```

| NAME | PLSQL_CCFLAGS |
|--------------|----------------|
| TEST_CCFLAGS | app_debug:TRUE |

Далее я изменяю настройки сеанса так, чтобы вычисление `$$app_debug` давало `FALSE`. Новая программа компилируется с этой настройкой:

```

SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:FALSE';
SQL> CREATE OR REPLACE PROCEDURE test_ccflags_new
2  IS
3  BEGIN
4      NULL;
5  END test_ccflags_new;
6  /

```

Затем существующая программа перекомпилируется с секцией REUSE SETTINGS:

```
SQL> ALTER PROCEDURE test_ccflags COMPILE REUSE SETTINGS;
```

Запрос к представлению словаря данных показывает, что теперь в разных программах используются разные настройки:

```
SQL> SELECT name, plsql_ccflags
2     FROM user_plsql_object_settings
3     WHERE NAME LIKE '%CCFLAGS%';
```

| NAME | PLSQL_CCFLAGS |
|------------------|-----------------|
| TEST_CCFLAGS | app_debug:TRUE |
| TEST_CCFLAGS_NEW | app_debug:FALSE |

Работа с обработанным кодом

Пакет DBMS_PREPROCESSOR используется для вывода или получения исходного текста вашей программы в *обработанной* (postprocessed) форме. DBMS_PREPROCESSOR предоставляет две программы, перегруженные для разных способов определения интересующего вас объекта, а также для использования отдельных строк и коллекций:

- DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE — получает обработанный исходный код и выводит его функцией DBMS_OUTPUT.PUT_LINE.
- DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE — возвращает обработанный исходный код в виде одной строки или коллекции строк.

При использовании версий этих программ, работающих с коллекциями, необходимо объявить, что коллекция базируется на следующей коллекции, определенной в пакете:

```
TYPE DBMS_PREPROCESSOR.source_lines_t IS TABLE OF VARCHAR2(32767)
INDEX BY BINARY_INTEGER;
```

Следующий фрагмент демонстрирует возможности этих программ. Сначала я компилирую очень маленькую программу с директивой выбора, проверяющей уровень оптимизации, а затем вывожу обработанный код с правильной ветвью команды \$IF:

```
/* Файл в Сети: cc_postprocessor.sql */
```

```
PROCEDURE post_processed
IS
BEGIN
$IF $$PLSQL_OPTIMIZE_LEVEL = 1
$THEN
    -- Медленно и просто
    NULL;
$ELSE
    -- Быстро, современно и просто
    NULL;
$END
END post_processed;
```

```
SQL> BEGIN
2     DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
3         'PROCEDURE', USER, 'POST_PROCESSED');
4 END;
5 /
```

```
PROCEDURE post_processed
IS
BEGIN
    -- Быстро, современно и просто
    NULL;
END post_processed;
```

В следующем блоке функция `get` используется для получения обработанного кода, а затем выводит его вызовом `DBMS_OUTPUT.PUT_LINE`:

```
DECLARE
    l_postproc_code    DBMS_PREPROCESSOR.SOURCE_LINES_T;
    l_row              PLS_INTEGER;
BEGIN
    l_postproc_code :=
        DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
            'PROCEDURE', USER, 'POST_PROCESSED');
    l_row := l_postproc_code.FIRST;

    WHILE (l_row IS NOT NULL)
    LOOP
        DBMS_OUTPUT.put_line ( LPAD (l_row, 3)
                               || ' - '
                               || rtrim ( l_postproc_code (l_row),chr(10))
                               );
        l_row := l_postproc_code.NEXT (l_row);
    END LOOP;
END;
```

Условная компиляция открывает массу полезных возможностей для разработчиков PL/SQL и администраторов приложений. Она становится еще более полезной с выходом новых версий базы данных Oracle и расширением набора констант `DBMS_DB_VERSION`, позволяющим более полно использовать особенности PL/SQL каждой версии.

PL/SQL и память экземпляров базы данных

Экономно используя машинные ресурсы (включая память и процессор), Oracle может поддерживать десятки тысяч одновременно работающих пользователей базы данных. Технологии управления памятью Oracle в последние годы стали довольно сложными и освоить их бывает непросто. И хотя средства автоматизации, появившиеся в последних версиях, основательно упрощают работу администраторов баз данных, разработчики PL/SQL должны понимать основные принципы управления памятью (особенно в том, что касается курсоров и пакетных переменных), чтобы избежать непроизводительных расходов памяти.

SGA, PGA и UGA

При взаимодействии клиентской программы (скажем, SQL*Plus или SQL Developer) с базой данных используются три структуры памяти.

SGA (System Global Area)

SGA представляет собой группу взаимосвязанных структур памяти, или *компонентов SGA*, содержащих данные и управляющую информацию по одному экземпляру Oracle Database. SGA совместно используется всеми серверными и фоновыми процессами. Примеры данных, хранящихся в SGA, — кэшированные блоки данных и общие области SQL.

PGA (Process Global Area)

PGA — область памяти, содержащая данные и управляющую информацию, которые монополично используются каждым серверным процессом. Oracle Database создает PGA при запуске процесса Oracle. Для каждого серверного и фонового процесса создается отдельная область PGA. Совокупность отдельных областей PGA называется *областью PGA экземпляра*. В параметрах инициализации базы данных задается размер области PGA экземпляра, но не отдельных PGA.

UGA (User Global Area)

Данные, которые должны сохраняться между обращениями к базе данных в ходе сеанса (пакетные переменные, приватные области SQL и т. д.), помещаются в UGA (User Global Area). По сути UGA используется для сохранения состояния сеанса.

Местонахождение UGA в памяти зависит от выбранного способа подключения к базе данных:

- **Выделенный сервер.** Для каждого сеанса Oracle создает выделенный серверный процесс. Такая конфигурация целесообразна при больших нагрузках, например при интенсивных вычислениях или долго выполняющихся запросах к базе данных. UGA помещается в PGA, потому что другим серверным процессам не нужно обращаться к этой области.
- **Общий сервер.** Обращения к базе данных ставятся в очередь к группе общих серверных процессов, которые могут обслуживать обращения от любых сеансов. Такая конфигурация хорошо подходит для сотен параллельных сеансов, генерирующих короткие обращения с относительно большим временем бездействия. UGA размещается в области SGA, чтобы данные были доступны для любого из общих серверных процессов.

Общий объем PGA существенно зависит от вида операций, выполняемых сервером для приложения. Например, для пакетов PL/SQL, заполняющих большие коллекции, может потребоваться большой объем памяти UGA.

Если приложение работает с общими серверами, пользовательским процессам приходится ждать своей очереди на обслуживание. Если ваш пользовательский процесс запускает долго выполняющиеся блоки PL/SQL или команды SQL, администратор базы данных должен либо настроить сервер с большим количеством теневых процессов, либо запустить эти сеансы на выделенном сервере.

А теперь давайте разберемся, как выглядит память с точки зрения выполняемой программы.

Курсоры и память

Возможно, вы уже написали сотни программ, которые объявляют и открывают курсоры, выбирают из них строки, а затем снова закрывают. Ни SQL, ни PL/SQL не могут работать без использования курсоров; многие операторы неявно выполняют рекурсивные вызовы, которые открывают дополнительные курсоры. А поскольку каждый курсор, явный или неявный, занимает память сервера базы данных, процесс оптимизации настройки Oracle часто подразумевает и сокращение количества необходимых приложению курсоров.



Хотя этот раздел посвящен управлению памятью, помните, что это всего лишь один из аспектов оптимизации базы данных; возможно, вам удастся повысить общую производительность, увеличивая количество курсоров.

Oracle ассоциирует курсоры с анонимными блоками PL/SQL практически так же, как с командами SQL. Например, при обработке первого вызова в текущем сеансе пользователя Oracle открывает в памяти UGA область («приватную» область SQL), в которой размещается информация, специфическая для этого вызова.

При выполнении команды SQL или блока PL/SQL сервер сначала проверяет, нет ли в содержимом библиотечного кэша готового представления этого кода. Если такая общая область PL/SQL будет найдена, исполняющее ядро связывает ее с приватной областью

SQL. Если такой области не существует, Oracle выполняет разбор команды или блока. (Кроме того, Oracle подготавливает и кэширует план выполнения анонимных блоков PL/SQL, включающий вызов ядра PL/SQL для интерпретации байт-кода.)

Oracle интерпретирует простейшие блоки PL/SQL (блоки, вызывающие подпрограммы, и блоки, не содержащие интегрированных команд SQL), используя только ту память, которая выделена для главного курсора. Если программа содержит вызовы PL/SQL или вызовы SQL, Oracle требуются дополнительные приватные области в памяти UGA. PL/SQL управляет ими от имени вашего приложения.

Мы подходим к важному факту, касающемуся работы с курсорами: существуют два способа закрытия курсора. Если это делается программно, то дальнейшее использование курсора в приложении без повторного открытия становится невозможным. Программное закрытие курсора выполняется командой `CLOSE`:

```
CLOSE имя_курсора;
```

или при автоматическом закрытии неявного курсора. Однако PL/SQL не сразу освобождает память, связанную с этим курсором, — курсор продолжает существовать как объект базы данных, чтобы избежать его обработки при возможном повторном открытии, как это часто случается. Заглянув в представление `V$OPEN_CURSOR`, вы увидите, что выполнение операции `CLOSE` не уменьшает количество открытых курсоров сеанса; в версии 11.1 также можно осуществить выборку из нового столбца `CURSOR_TYPE` для получения дополнительной информации о курсоре.

PL/SQL поддерживает собственный «сеансовый кэш курсоров», то есть сам решает, когда какой курсор необходимо освободить. Максимальное количество курсоров в том кэше задается инициализационным параметром `OPEN_CURSORS`. Выбор курсора для освобождения памяти производится на основании алгоритма LRU (Least Recently Used — «дольше всех не использовавшийся»). Однако внутренний алгоритм PL/SQL работает оптимально только в том случае, если все курсоры закрываются сразу же после завершения выборки данных. Итак, запомните: курсоры, явно открываемые в программе, следует *явно закрывать* сразу же после завершения использования.

Внутренний алгоритм работает оптимально только в том случае, если ваши программы закрывают курсоры сразу же после завершения выборки. Итак, запомните:

Если программа явно открывает курсор, всегда явно закрывайте его сразу же после завершения работы (но не ранее)!

У программиста имеется возможность вмешаться в стандартное поведение Oracle. Конечно, для закрытия всех курсоров сеанса можно завершить сам сеанс! Существуют и другие, менее радикальные способы:

- сброс состояния пакета (см. ниже «Большие коллекции в PL/SQL»);
- низкоуровневое управление поведением курсоров с помощью пакета `DBMS_SQL` (правда, выигрыш от использования данного подхода может быть значительно меньше, чем потери от снижения производительности и усложнения программирования).

Советы по экономии памяти

Разобравшись с теорией, мы переходим к практическим советам, которыми вы сможете воспользоваться в повседневной работе. Также обратите внимание на более общие рекомендации по оптимизации программ из главы 21. Кроме того, полезно иметь возможность измерить объем памяти, используемой сеансом в любой момент времени, из кода приложения. Для этого можно выдать запрос к различным представлениям `v$`. Пакет `plsql_memory` (см. файл `plsql_memory.pkg` на сайте книги) поможет вам в этом.

Совместное использование команд

База данных способна предоставить программам совместный доступ к откомпилированным версиям команд SQL и анонимным блокам даже в том случае, если они получены от разных сеансов и разных пользователей. Оптимизатор определяет план выполнения во время разбора, поэтому факторы, влияющие на разбор (в том числе и настройки оптимизатора), повлияют на совместное использование команд SQL. Чтобы система могла обеспечить совместный доступ к командам SQL, необходимо придерживаться нескольких основных правил:

- Значения переменных должны задаваться посредством переменных привязки, а не в виде литералов, чтобы текст инструкций оставался неизменным. Сами переменные привязки должны иметь соответствующие имена и типы данных.
- Правила регистра символов и форматирования в исходном коде должны точно совпадать. Если вы выполняете одни и те же программы, это произойдет автоматически. «Одноразовые» команды могут не совпадать с командами из программ на 100%.
- Ссылки на объекты базы данных должны разрешаться как ссылки на один и тот же объект.
- Для SQL параметры базы данных, влияющие на работу оптимизатора запросов SQL, должны совпадать. Например, в вызывающих сеансах необходимо задавать одинаковый критерий оптимизации (ALL_ROWS или FIRST_ROWS).
- Вызывающие сеансы должны поддерживать одни и те же национальные языки (National Language Support, NLS).

Мы не станем останавливаться на двух последних правилах; конкретные причины, препятствующие совместному использованию команд SQL, можно найти в представлении `v$sql_shared_cursor`. Сейчас нас интересует в первую очередь влияние первых трех правил для программ PL/SQL.

Первое правило (привязка) настолько критично, что ему посвящен отдельный подраздел.

Второе правило (регистр и форматирование) является хорошо известным условием совместного использования инструкций. Текст должен точно совпадать, потому что база данных вычисляет по нему хеш-код для поиска и блокировки объекта в кэше библиотек.

Несмотря на то что PL/SQL обычно игнорирует регистр, следующие три блока не воспринимаются как одинаковые:

```
BEGIN NULL; END;  
begin null; end;  
BEGIN NULL;   END;
```

Для них генерируются разные хеш-коды, поэтому инструкции считаются разными — на логическом уровне они эквиваленты, но на физическом уровне они различаются. Однако если все ваши анонимные блоки коротки, а все «настоящие программы» реализованы в виде хранимого кода (например, в виде пакетов), вероятность ошибочно подавить их совместное использование значительно меньше.

Реализуйте код SQL и PL/SQL в виде хранимых программ. Анонимные блоки должны быть как можно короче, в общем случае — состоять из единственного вызова хранимой программы. Кроме того, можно дать еще одну рекомендацию, относящуюся к SQL: чтобы обеспечить возможность совместного использования SQL-инструкций, помещайте их в программы, которые вызываются из разных мест приложения. Это избавит вас от необходимости по несколько раз писать одни и те же инструкции.

Третье правило гласит, что внешние ссылки (на таблицы, процедуры и т. д.) должны разрешаться как ссылки на один и тот же объект. Допустим, мы с пользователем `Scott` подключились к Oracle, и оба запустили такой блок:

```
BEGIN
  XYZ;
END;
```

Решение Oracle о том, следует ли вам обоим предоставить возможность использования кэшируемой формы данного блока, зависит от того, ссылается ли имя «xyz» на одну и ту же хранимую процедуру. Если пользователь Scott определил синоним xyz, указывающий на нашу копию процедуры, то Oracle разрешит совместный доступ к анонимному блоку. При наличии независимых копий процедуры каждый будет работать со своим блоком. И даже если обе копии процедуры xyz будут абсолютно идентичны, Oracle будет кэшировать их как разные объекты. Аналогичным образом как разные объекты кэшируются идентичные триггеры разных таблиц. Из сказанного можно сделать вывод: избегайте создания одинаковых копий таблиц и программ под разными учетными записями.

Согласно общепринятому мнению, для экономии памяти нужно отделить программный код, общий для нескольких программ (и в особенности триггеров), и реализовать его в отдельном вызове. Иначе говоря, одна учетная запись базы данных назначается владельцем программ PL/SQL, а другим пользователям, которым эти программы нужны, предоставляются привилегии EXECUTE. Хотя эта практика очень хорошо влияет на удобство сопровождения, вряд ли она обеспечит реальную экономию памяти. Более того, для каждой вызывающей стороны создается новый объект с дополнительными затратами памяти в несколько килобайтов за сеанс. Конечно, значительные затраты памяти будут наблюдаться только при очень большом количестве пользователей.

У устоявшихся стереотипов есть еще одна проблема, проявляющаяся в среде с множеством пользователей, одновременно выполняющих одну программу PL/SQL. При вызове общего кода необходим механизм установления и снятия блокировки библиотечного кода, который может привести к задержкам. В таких случаях дублирование кода может быть предпочтительным, потому что оно предотвращает установление лишних блокировок и сокращает риск снижения производительности.

Но давайте вернемся к первому правилу, касающемуся переменных привязки.

Переменные привязки

В среде Oracle *переменной привязки* называется входная переменная в команде, значение которой передается из среды вызывающей стороны. Переменные привязки играют особенно важную роль в совместном использовании команд SQL независимо от источника инструкций: PL/SQL, Java, SQL*Plus или OCI. Переменные привязки упрощают масштабирование приложений, помогают бороться с внедрением кода (code injection) и способствуют совместному использованию команд SQL.

Чтобы две инструкции считались идентичными, использованные в них переменные привязки должны совпадать по имени, типу данных и максимальной длине. Например, следующие две инструкции идентичными *не считаются*:

```
SELECT col FROM tab1 WHERE col = :bind1;
SELECT col FROM tab1 WHERE col = :bind_1;
```

Однако это требование относится лишь к тексту инструкции, воспринимаемому ядром SQL. Как упоминалось ранее в этой главе, PL/SQL переформулирует статические инструкции SQL еще до того, как SQL их увидит. Пример:

```
FUNCTION psql_bookcount (author IN VARCHAR2)
  RETURN NUMBER
IS
  titlepattern VARCHAR2(10) := '%PL/SQL%';
  lcount NUMBER;
```

продолжение ➤

```

BEGIN
  SELECT COUNT(*) INTO lcount
    FROM books
   WHERE title LIKE titlepattern
     AND author = plsql_bookcount.author;
  RETURN lcount;
END;

```

После выполнения `plsql_bookcount` представление `V$SQLAREA` в Oracle11g показывает, что PL/SQL переформулировал запрос в следующем виде:

```
SELECT COUNT(*) FROM BOOKS WHERE TITLE LIKE :B2 AND AUTHOR = :B1
```

Параметр `author` и локальная переменная `titlepattern` заменены переменными привязки `:B1` и `:B2`. Таким образом, в статическом коде SQL вам не нужно беспокоиться о соответствии имен переменных привязки; PL/SQL заменяет имя переменной сгенерированным именем переменной привязки.

Автоматическое введение переменных привязки в PL/SQL распространяется на переменные, используемые в предложениях `WHERE` и `VALUES` статических инструкций `INSERT`, `UPDATE`, `MERGE`, `DELETE` и, конечно, `SELECT`.

Дополнительные эксперименты показали, что изменение максимальной длины переменной PL/SQL не приводило к появлению дополнительной инструкции в области SQL, но с изменением типа данных переменной такая инструкция появляется. Впрочем, я не прошу верить мне на слово; при наличии необходимых привилегий вы можете провести собственные эксперименты и определить, действительно ли команды SQL совместно используются в соответствии с вашими планами. Загляните в представление `V$SQLAREA`. Результат выборки для приведенного выше кода:

```

SQL> SELECT executions, sql_text
      2 FROM v$sqlarea
      3 WHERE sql_text like 'SELECT COUNT(*) FROM BOOKS%'

```

```
EXECUTIONS SQL_TEXT
```

```

-----
      1 SELECT COUNT(*) FROM BOOKS WHERE TITLE LIKE :B2
        AND AUTHOR = :B1

```

Если PL/SQL настолько умен, значит, можно не беспокоиться о переменных привязки? Не торопитесь: хотя PL/SQL автоматически осуществляет привязку переменных в статических SQL-инструкциях, эта возможность недоступна в *динамическом SQL*. Неаккуратное программирование легко приведет к формированию команд с литералами. Пример:

```

FUNCTION count_recent_records (tablename_in IN VARCHAR2,
  since_in IN VARCHAR2)
RETURN PLS_INTEGER
AS
  count_1 PLS_INTEGER;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '
    || DBMS_ASSERT.SIMPLE_SQL_NAME(tablename_in)
    || ' WHERE lastupdate > TO_DATE('
    || DBMS_ASSERT.ENQUOTE_LITERAL(since_in)
    || ', ' || 'YYYYMMDD')'
    INTO count_1;
  RETURN count_1;
END;

```

При его выполнении динамически строятся команды следующего вида::

```
SELECT COUNT(*) FROM имя_таблицы WHERE lastupdate > TO_DATE('20090315', 'YYYYMMDD')
```

Повторные вызовы с разными аргументами `since_in` могут привести к генерированию большого количества инструкций, которые вряд ли будут использоваться совместно:

```
SELECT COUNT(*) FROM tablename WHERE lastupdate > TO_DATE('20090105', 'YYYYMMDD')
SELECT COUNT(*) FROM tablename WHERE lastupdate > TO_DATE('20080704', 'YYYYMMDD')
SELECT COUNT(*) FROM tablename WHERE lastupdate > TO_DATE('20090101', 'YYYYMMDD')
```

Естественно, это приводит к весьма расточительному расходованию памяти и других серверных ресурсов.

ИСПОЛЬЗОВАНИЕ DBMS_ASSERT ДЛЯ БОРЬБЫ С ВНЕДРЕНИЕМ КОДА

Что это за вызовы DBMS_ASSERT в примере с переменными привязки? Динамический SQL, который использует данные, непосредственно вводимые пользователем, следует проверить, прежде чем без оглядки выполнять их. Вызов DBMS_ASSERT помогает убедиться в том, что код получает именно те данные, которые он рассчитывает получить. Если попытаться вызвать функцию `count_recent_records` для «странного» имени таблицы вида «books where 1=1;--», DBMS_ASSERT выдаст исключение и остановит программу еще до того, как она успеет причинить вред. DBMS_ASSERT.SIMPLE_SQL_NAME гарантирует, что входные данные соответствуют критериям действительного имени SQL. DBMS_ASSERT.ENQUOTE_LITERAL заключает входные данные в кавычки и проверяет, что они не содержат встроенных кавычек. Полное описание DBMS_ASSERT приведено в документации Oracle PL/SQL Packages and Types Reference.

Если переписать эту же функцию с использованием переменной привязки, получится:

```
FUNCTION count_recent_records (tablename_in IN VARCHAR2,
    since_in IN VARCHAR2)
RETURN PLS_INTEGER
AS
    count_1 PLS_INTEGER;
BEGIN
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '
        || DBMS_ASSERT.SIMPLE_SQL_NAME(tablename_in)
        || ' WHERE lastupdate > :thedate'
        INTO count_1
        USING TO_DATE(since_in, 'YYYYMMDD');
    RETURN count_1;
END;
```

Компилятор SQL получит инструкции следующего вида:

```
SELECT COUNT(*) FROM имя_таблицы WHERE lastupdate > :thedate
```

Вторая версия не только проще и понятнее, но и обеспечивает значительно лучшую производительность при повторных вызовах с одинаковыми значениями аргумента `tablename_in` при разных значениях `since_in`.

Oracle также поддерживает параметр инициализации `CURSOR_SHARING`, который *может* предоставить некоторые преимущества в приложениях с большим объемом кода без переменных привязки. Присваивая этому параметру значение `FORCE` или `SIMILAR`, можно потребовать, чтобы база данных заменяла литералы SQL (полностью или частично) переменными привязки, предотвращая затраты на разбор. К сожалению, это одна из тех возможностей, которые в теории работают лучше, чем на практике.

С другой стороны, если вы будете внимательно относиться к использованию полноценных переменных привязки в динамическом коде SQL, ваши усилия будут вознаграждены

во время выполнения (только не забудьте оставить параметру `CURSOR_SHARING` значение по умолчанию `EXACT`).



Даже если вам удастся добиться некоторого повышения производительности за счет использования `CURSOR_SHARING`, рассматривайте этот способ как полумеру. По эффективности его даже нельзя сравнивать с полноценными переменными привязки, к тому же он может порождать ряд непредвиденных и нежелательных побочных эффектов. Если вам приходится использовать эту возможность из-за аномалий программных продуктов (часто сторонних), делайте это только до тех пор, пока не удастся модифицировать код для перехода на полноценные переменные привязки. Также учтите, что соответствующую настройку можно включить на уровне сеанса при помощи триггера `LOGON`.

Пакеты и эффективное использование памяти

При извлечении байт-кода хранимой программы PL/SQL читается вся программа. Речь идет не только о процедурах и функциях, но и о пакетах базы данных. Иначе говоря, нельзя сделать так, чтобы считывалась лишь часть пакета, — при обращении к любому элементу, даже к одной-единственной переменной, в библиотечный кэш загружается весь откомпилированный код пакета. Поэтому деление кода на меньшее количество пакетов большего размера приводит к меньшим затратам памяти (и дискового пространства), чем с множеством мелких пакетов. Таким образом, логическая группировка элементов пакета полезна не только с точки зрения архитектуры, но и с точки зрения производительности системы.



Поскольку Oracle считывает в память весь пакет, объединять в пакеты следует только функционально связанные элементы, то есть такие, которые с большой вероятностью будут вызываться в одном сеансе.

Большие коллекции в PL/SQL

Совместное использование объектов — прекрасное решение, но допустимо оно далеко не для всех объектов программы. Даже если несколько пользователей выполняют одну и ту же программу, принадлежащую одной схеме Oracle, каждый сеанс имеет собственную область памяти, в которой содержатся специфические для этого вызова данные — значения локальных и пакетных переменных, константы и курсоры. И пытаться организовать совместное использование этих данных, относящихся к конкретному сеансу, бессмысленно. Наиболее типичные проблемы возникают при работе с коллекциями (о коллекциях подробно рассказывалось в главе 12). Предположим, ассоциативный массив PL/SQL объявляется в программе следующим образом:

```
DECLARE
  TYPE number_tab_t IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  number_tab number_tab_t;
  empty_tab number_tab_t;
```

В массив включается большое количество элементов:

```
FOR i IN 1..100000
LOOP
  number_tab(i) := i;
END LOOP;
```

Все эти элементы должны где-то храниться. По приведенным выше правилам, память для этого массива будет выделена в глобальной области пользователя UGA, если он

объявлен на уровне пакета, или в глобальной области вызова CGA, если это данные анонимного блока, процедуры или функции верхнего уровня. В любом случае для работы с большой коллекцией потребуется очень большой объем памяти.

Возникает вопрос, как освободить эту память после завершения работы с коллекцией? Нужно выполнить одну из двух команд:

```
number_tab.DELETE;
```

или

```
number_tab := empty_tab;
```

В любом из этих случаев Oracle освободит память в своем списке свободных объектов. Это означает, что память для хранения переменной пакета будет возвращена в динамический пул состояния сеанса, а память переменной уровня вызова — в CGA. То же произойдет и в том случае, если переменная, объявленная как коллекция, выйдет за пределы области видимости. Например, если коллекция будет объявлена в отдельной процедуре, Oracle освободит занимаемую ею память сразу после завершения работы процедуры. В любом случае такая память не будет доступна ни другим сеансам, ни текущему сеансу, если ему понадобится память из области CGA. Если в последующих операциях DML будет, допустим, выполняться сортировка большого объема данных, приложению может потребоваться огромный объем памяти. И только после завершения сеанса память будет полностью освобождена и вернется в родительский динамический пул (кучу).

Следует подчеркнуть, что для системы управления виртуальной памятью не составит труда управлять файлом подкачки большого объема, особенно если процесс удерживает в своем адресном пространстве большое количество неактивной виртуальной памяти. Эта неактивная память занимает только место на жестком диске, а не реальную память. Однако иногда бывает нежелательно заполнять страничное пространство, поэтому будет лучше, если Oracle освободит память. Для таких случаев существует специальная процедура «уборки мусора» с очень простым синтаксисом:

```
DBMS_SESSION.FREE_UNUSED_USER_MEMORY;
```

Эта встроенная процедура найдет большую часть памяти UGA, не используемой переменными программы, и вернет ее в родительскую кучу: при наличии выделенного сервера — в PGA, а в случае общего сервера — в SGA.

Я основательно протестировал процесс освобождения занимаемой коллекциями памяти в разных ситуациях, в частности при использовании ассоциативных массивов и вложенных таблиц для выделенного и общего сервера, анонимных блоков и данных пакетов. В результате были сделаны следующие выводы:

- Для освобождения занимаемой памяти недостаточно присвоить вложенной таблице или массиву `VARRAY` значение `NULL`. Нужно либо вызвать метод коллекции `DELETE`, либо присвоить коллекции другую пустую, но инициализированную коллекцию, либо подождать, пока она выйдет из области видимости.
- Чтобы освободить память и вернуть ее в родительскую кучу, используйте процедуру `DBMS_SESSION.FREE_UNUSED_USER_MEMORY`, когда ваша программа заполнила одну или несколько больших таблиц PL/SQL, пометила их как неиспользуемые, и скорее всего, ей не придется выделять крупные блоки памяти для аналогичных операций.
- В режиме общего сервера ошибки нехватки памяти обычно происходят чаще, чем в режиме выделенного сервера, поскольку область UGA выделяется из системной глобальной области SGA, имеющей ограниченный размер. Как указано в разделе «Что делать при нехватке памяти» (см. далее), при этом может произойти ошибка ORA-04031.
- В режиме общего сервера освободить память, которую занимают таблицы PL/SQL, невозможно (если только таблица не объявлена на уровне пакета).

На практике объем памяти, занимаемой коллекцией элементов типа **NUMBER**, не зависит от того, содержат ли элементы значения **NULL** или, допустим, 38-значные числа. А вот для значений типа **VARCHAR2**, объявленных с длиной более 30 символов, Oracle, похоже, выделяет память динамически.

При заполнении ассоциативного массива в режиме выделенного сервера массив, содержащий миллион значений типа **NUMBER**, занимает около 38 Мбайт. И даже если элементы массива имеют тип **BOOLEAN**, Oracle9i использует для него почти 15 Мбайт памяти. Умножьте это значение на количество пользователей, например на 100, — результат будет огромным, особенно если выгрузка памяти на диск нежелательна по соображениям производительности.

Чтобы узнать, сколько памяти UGA и PGA использует текущий сеанс, выполните запрос следующего вида:

```
SELECT n.name, ROUND(m.value/1024) kbytes
  FROM V$STATNAME n, V$MYSTAT m
 WHERE n.statistic# = m.statistic#
        AND n.name LIKE 'session%memory%'
```

(Для чтения представлений из этого примера недостаточно привилегий по умолчанию.) Запрос выдаст текущие и максимальные затраты памяти в сеансе.

Если вы хотите освободить память, используемую пакетными коллекциями без завершения сеанса, вызовите одну из двух встроенных процедур.

- **DBMS_SESSION.RESET_PACKAGE** — освобождает всю память, выделенную для хранения информации о состоянии пакета. В результате все переменные пакета получают значения по умолчанию. Для пакетов эта встроенная процедура делает больше, чем пакет **FREE_UNUSED_USER_MEMORY**, поскольку она не обращает внимания на то, используется память или нет.
- **DBMS_SESSION.MODIFY_PACKAGE_STATE** (*флаги_операций* IN **PLS_INTEGER**) — в параметре *флаги_операций* можно задать одну из двух констант: **DBMS_SESSION.free_all_resources** или **DBMS_SESSION.reinitialize**. Использование первой приводит к тому же эффекту, что и применение процедуры **RESET_PACKAGE**. Вторая константа восстанавливает переменные состояния, присваивая им значения по умолчанию, но не освобождает и не воссоздает данные пакета с нуля. Кроме того, она программно закрывает открытые курсоры и не очищает их кэш. Если в вашей ситуации это приемлемо, используйте вторую константу, потому что по скорости она превосходит полный сброс пакета.

Операции **BULK COLLECT...LIMIT**

BULK-операции повышают эффективность обработки данных, но вы должны следить за тем, чтобы затраты памяти оставались умеренными, а коллекции не росли до слишком больших размеров. При выборке **BULK COLLECT** в коллекцию по умолчанию загружаются все строки. При больших объемах данных коллекция получается слишком большой. В таких случаях на помощь приходит конструкция **LIMIT**.

В ходе тестирования выяснилось, что она сокращает затраты памяти, что было ожидаемо, но оказалось, что оно еще и ускоряет работу программ. В следующем примере используется тестовая таблица с миллионом строк. Чтобы получить надежные данные для сравнения, программа сначала была выполнена для предварительного заполнения кэша, а затем она была запущена снова после повторного подключения к базе данных. Для вывода информации об использовании памяти использовались средства пакета **plsql_memory** (см. файл **plsql_memory.pkg** на сайте книги):


```

/* Файл в Сети: LimitBulkCollect.sql */
DECLARE
  -- Подготовка коллекций
  TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  TYPE nametab IS TABLE OF VARCHAR2(4000) INDEX BY PLS_INTEGER;
  TYPE tstab IS TABLE OF TIMESTAMP INDEX BY PLS_INTEGER;
  CURSOR test_c IS
    SELECT hi_card_nbr,hi_card_str ,hi_card_ts
    FROM data_test
  ;
  nbrs numtab;
  txt nametab;
  tstamps tstab;
  counter number;
  strt number;
  fnsh number;
BEGIN
  plsql_memory.start_analysis; -- Инициализация вывода данных о затратах памяти
  strt := dbms_utility.get_time; -- Сохранение начального времени
  OPEN test_c;
  LOOP
    FETCH test_c BULK COLLECT INTO nbrs,txt,tstamps LIMIT 10000;
    EXIT WHEN nbrs.COUNT = 0;
    FOR i IN 1..nbrs.COUNT LOOP
      counter := counter + i; -- Обработка данных
    END LOOP;
  END LOOP;
  plsql_memory.show_memory_usage;
  CLOSE test_c;
  fnsh := dbms_utility.get_time;
  -- Преобразование сотых долей секунды в миллисекунды
  DBMS_OUTPUT.PUT_LINE('Run time = '||((fnsh-strt)*100)||' ms');
END;
/

```

Результаты, которые я получил:

```

Change in UGA memory: 394272 (Current = 2459840)
Change in PGA memory: 1638400 (Current = 5807624)
Run time = 1530 ms

```

Мы видим, что с пределом в 10 000 записей затраты памяти PGA выросли на 1 638 400 байт. При повторном выводе информации после завершения блока PL/SQL бóльшая часть этой памяти (хотя и не вся) освобождается:

```

EXEC plsql_memory.show_memory_usage;
Change in UGA memory: 0 (Current =2394352)
Change in PGA memory: -458752 (Current = 3907080)

```

Затем было проведено повторное тестирование без LIMIT, чтобы все строки таблицы загружались за один раз:

```

Change in UGA memory: 0 (Current = 1366000)
Change in PGA memory: 18153472 (Current = 22519304)

```

Как видите, без LIMIT используется намного больше памяти. Итак, я настоятельно рекомендую включать LIMIT в окончательную версию ваших приложений.

Сохранение состояния объектов

Обычно Oracle сохраняет до окончания сеанса значения констант уровня пакета, переменных и состояние курсора в области UGA. С переменными, объявленными в разделе объявлений отдельного модуля, дело обстоит иначе. Поскольку их область видимости ограничена модулем, по завершении его работы занимаемая данными память освобождается. С этого момента их самих больше не существует.

Кроме отключения от сервера базы данных, потеря состояния пакета может происходить еще по нескольким причинам:

- программа была перекомпилирована или по иной причине стала недействительной с точки зрения базы данных;
- в текущем сеансе была выполнена встроенная процедура `DBMS_SESSION.RESET_PACKAGE`;
- в код программы добавлена директива компилятора `SERIALLY_REUSABLE` (см. главу 18), предписывающая Oracle сохранять параметры состояния только на время вызова, а не в течение всего сеанса;
- в программе используется веб-шлюз в режиме по умолчанию, в котором параметры сеанса каждого клиента не сохраняются;
- в клиентский сеанс передается такая ошибка, как `ORA-04069` (не удастся удалить или заменить библиотеку с зависимой таблицей).

С учетом этих ограничений структуры данных пакета могут действовать в среде PL/SQL как глобальные. Иначе говоря, они могут использоваться программами PL/SQL, выполняемыми в одном сеансе, для обмена данными.

С точки зрения архитектуры приложения глобальные данные делятся на два вида: открытые и приватные.

- **Открытые данные.** Структура данных, объявленная в спецификации пакета, является глобальной открытой структурой данных. Доступ к ней имеют любые программы и пользователи с привилегией `EXECUTE`. Программы даже могут присваивать произвольные значения пакетным переменным, которые не объявлены как константы. Как известно, открытые глобальные данные являются потенциальной причиной многих ошибок: их удобно объявлять, но применение их «на скорую руку» порождает неструктурированный код с опасными побочными эффектами.
- **Приватные данные.** Приватные глобальные структуры данных не вызывают столько проблем. В спецификации пакета они отсутствуют и извне на них ссылаться нельзя. Эти данные предназначены для использования только внутри пакета и только его элементами.



Пакетные данные глобальны только в рамках одного сеанса или подключения к базе данных. Они не используются совместно несколькими сеансами. Если вам потребуется организовать совместный доступ к данным между сеансами, для этого существуют другие средства: пакет `DBMS_PIPE` package, Oracle Advanced Queuing, пакет `UTL_TCP`... не говоря уже о таблицах базы данных!

Что делать при нехватке памяти

Предположим, вы работаете с базой данных; все идет прекрасно, выполняется множество команд SQL и PL/SQL, и вдруг как гром с ясного неба: ошибка `ORA-04031`, не удастся выделить *n* байт общей памяти. Такая ошибка чаще встречается в режиме общего сервера с его повышенным расходом памяти UGA общего сервера. В режиме выделенного сервера база данных обычно может получить больше виртуальной памяти от операционной системы, но и в этом случае может возникнуть аналогичная ошибка `ORA-04030`. Собственно, с выделенным сервером максимальный объем памяти составляет примерно 4 Гбайт на сеанс, тогда как с общим сервером можно вручную назначить пулу любой нужный размер.

Исправить положение можно несколькими способами. Если вы являетесь разработчиком приложения, попробуйте сократить использование общей памяти. Некоторые возможные действия (приблизительно в порядке применения):

1. Внесите изменения в код и обеспечьте совместное использование максимального количества команд SQL.
2. Сократите размер или количество коллекций, хранящихся в памяти.
3. Сократите объем кода приложения в памяти.
4. Измените настройки уровня базы данных и/или купите дополнительную память для сервера.

Пункты 1 и 2 уже были рассмотрены; рассмотрим пункт 3. Как оценить размер исходного кода после того, как он будет загружен в память? И как сократить его?

Прежде чем запускать программу PL/SQL, база данных должна загрузить в память весь свой байт-код. Чтобы узнать, сколько места занимает программный объект в общем пуле, попросите своего администратора базы данных выполнить встроенную процедуру `DBMS_SHARED_POOL.SIZES`, которая перечисляет все объекты с размером больше заданного.

Следующий пример выводит информацию о затратах памяти, необходимых объектам в общем пуле сразу же после запуска базы данных¹:

```
SQL> SET SERVEROUTPUT ON SIZE 1000000
SQL> EXEC DBMS_SHARED_POOL.sizes(minsize => 125)
```

```
SIZE(K) KEPT NAME
-----
433      SYS.STANDARD (PACKAGE)
364      SYS.DBMS_RCVMAN (PACKAGE BODY)
249      SYSMAN.MGMT_JOB_ENGINE (PACKAGE BODY)
224      SYS.DBMS_RCVMAN (PACKAGE)
221      SYS.DBMS_STATS_INTERNAL (PACKAGE)
220      SYS.DBMS_BACKUP_RESTORE (PACKAGE)
125      MERGE INTO cache_stats_1$ D USING (select * from table(dbms_sta
ts_internal.format_cache_rows(CURSOR((select dataobj# o, st
atic# stat, nvl(value, 0) val from gv$segstat where stat
istic# in (0, 3, 5) and obj# > 0 and inst_id = 1) union all
(select obj# o, 7 stat,nvl(sum(num_buf), 0) val from x$kcb
oqh x where inst_id = 1 group by obj#) order by o))) wh
(20B5C934,3478682418) (CURSOR)
```

Условие `minsize => 125` означает «выводить только объекты с размером 125 Кбайт и выше». Из выходных данных видно, что больше всего общей памяти (433 Кбайт) занимает пакет `STANDARD`².

Если вы хотите избавиться от ошибок 4031 или 4030, знать объем памяти, используемой программами, необходимо, но не достаточно; также нужно знать размер общего пула и объем памяти, занимаемой «воссоздаваемыми» объектами — то есть объектами, которые могут устаревать, вытесняться из памяти и загружаться повторно при необходимости. Некоторые из этих сведений трудно получить от базы данных; вам может потребоваться знание таинственных представлений `x$`. Тем не менее версии 9.2.0.5 и выше автоматически генерируют дампы кучи в каталоге `USER_DUMP_DEST` при возникновении ошибки 4031. Посмотрите, что можно извлечь из полученной информации, или просто передайте ее в техническую поддержку Oracle. Также попробуйте определить, не содержит ли приложение большой объем неразделяемого кода, который логичнее было бы сделать общим, потому что это может оказать большое влияние на затраты памяти.

¹ Почему столбцы данных неправильно выравниваются по своим заголовкам? Вероятно, из-за серьезных ограничений `DBMS_OUTPUT`. Если вас это не устраивает, напишите собственную реализацию (запросите данные из `V$SQLAREA` после выполнения пакета).

² Старые версии `DBMS_SHARED_POOL.SIZES` содержали ошибку, из-за которой выводимые результаты были завышены приблизительно на 2.3%. Пакет Oracle ошибочно вычислял размер в килобайтах делением на 1000 вместо 1024.

Программы PL/SQL, откомпилированные в низкоуровневый код, компонируются в общие библиотечные файлы, но база данных все равно выделяет некоторую память для их выполнения. Привилегированный пользователь может использовать средства операционной системы (такие, как `rmap` в Solaris) для изменения объема памяти, занимаемой за пределами базы данных.

Теперь обратимся к шагу 4 — настройке базы данных или приобретению дополнительной памяти. Компетентный администратор базы данных знает, как настроить общий пул при помощи следующих параметров:

- `SHARED_POOL_SIZE` — байты, зарезервированные для общего пула.
- `DB_CACHE_SIZE` — байты памяти, зарезервированные для хранения строк данных из базы данных (возможно, вам придется сократить это значение для увеличения размера общего пула).
- `LARGE_POOL_SIZE` — байты памяти, зарезервированные для необязательного блока, в котором хранится область UGA подключений к общему серверу (предотвращает конкуренцию за использование общего пула со стороны переменной части UGA).
- `JAVA_POOL_SIZE` — байты, используемые менеджером памяти Java.
- `STREAMS_POOL_SIZE` — байты, используемые технологией Oracle Streams.
- `SGA_TARGET` — размер области SGA, из которой база данных будет автоматически выделять упоминавшиеся выше кэш и пул (ненулевое количество байтов).
- `PGAAggregate_TARGET` — общий объем памяти, используемой всеми серверными процессами в экземпляре. Обычно равен объему серверной памяти, доступной для базы данных, за вычетом размера SGA.
- `PGAAggregate_LIMIT` (появился в Oracle Database 12c) — задает ограничение агрегатной памяти PGA, потребляемой экземпляром. При превышении лимита вызовы сеансов, использующие больше всего памяти, будут отменены. Параллельные запросы будут рассматриваться как одно целое. Если суммарное использование памяти PGA по-прежнему превышает лимит, то сеансы с наибольшими затратами памяти завершаются. Описанные действия не применяются к процессам `SYS` и критичным фоновым процессам.

Вы также можете попросить своего администратора обеспечить принудительное хранение в памяти программ PL/SQL, последовательностей, таблиц или курсоров при помощи процедуры `DBMS_SHARED_POOL.KEEP`¹.

Например, следующий блок требует, чтобы база данных зафиксировала пакет `STANDARD` в памяти:

```
BEGIN
    DBMS_SHARED_POOL.KEEP('SYS.STANDARD');
END;
```

Фиксировать в памяти особенно полезно большие программы, которые выполняются относительно редко. Без фиксации частично откомпилированный код с большой вероятностью будет вытеснен из-за долгого бездействия, и при повторном вызове его загрузка может привести к вытеснению из пула множества меньших объектов (и соответствующим потерям производительности).

Возможно, следующий совет покажется очевидным, но если у небольшого подмножества пользователей или приложений возникают ошибки `ORA-04031` — попробуйте перевести «нарушителей» в режим выделенного сервера.

¹ В документации Oracle мелким шрифтом написано, что эта процедура может устареть с появлением улучшенных алгоритмов управления памятью.

Компиляция в низкоуровневый код

В режиме по умолчанию (интерпретация) код частично компилируется, а частично интерпретируется во время выполнения. PL/SQL выполняется в виртуальной машине; сначала код транслируется в виртуальный машинный код, иногда называемый *байт-кодом* или *m-кодом*. По сути происходит то же самое, что и при выполнении программ Java. Когда приходит время выполнять код, байт-код транслируется (интерпретируется) в вызовы системных функций.

Но когда код начнет нормально работать, для повышения эффективности программ PL/SQL можно приказать базе данных преобразовать байт-код в машинный код на более ранней стадии, во время компиляции (так называемый *низкоуровневый режим*). В результате база данных будет динамически загружать откомпилированный машинный код во время выполнения.

Когда используется режим интерпретации

Итак, если низкоуровневый режим работает быстрее, зачем выполнять программы в режиме интерпретации? Попробуем взглянуть на этот вопрос под другим углом. Целью низкоуровневого режима является высокая скорость выполнения. Таким образом, чтобы добиться максимальной скорости, вы поднимаете уровень оптимизации и пытаетесь выполнить как можно большую часть работы (включая раннее преобразование в машинный код) до стадии выполнения. В ходе разработки и модульного тестирования кода средства отладки важнее высокой скорости выполнения. Если вам понадобится выполнить исходный код в пошаговом режиме с заходом в подпрограммы, вас наверняка не устроит перестановка исходного кода оптимизирующим компилятором (уровень оптимизации 2) или подстановка подпрограмм (уровень оптимизации 3). Итак, во время отладки следует использовать уровень оптимизации 0 или 1; на этой стадии полезность низкоуровневого режима уже не так очевидна. Я рекомендую применять выполнение в режиме интерпретации в среде разработки.

Когда используется низкоуровневый режим

Низкоуровневый режим ориентирован на скорость. Программа запускается в низкоуровневом режиме, когда она уже отлажена и должна работать с максимальной скоростью. Компиляция в низкоуровневый код идет рука об руку с более высокими уровнями оптимизации. Обычно эта конфигурация применяется для окончательных версий программ, а также в некоторых тестовых средах. В низкоуровневом режиме время компиляции слегка увеличивается, потому что компилятору приходится выполнять больше работы, но программа и выполняется быстрее, чем в режиме интерпретации (по крайней мере не медленнее).

Обратите внимание: компиляция PL/SQL предоставляет наименьший выигрыш в производительности подпрограмм, основное время которых тратится на выполнение SQL. Кроме того, при одновременном выполнении многих подпрограмм, откомпилированных в низкоуровневом режиме, потребуется выделить большой объем общей памяти, что может повлиять на производительность системы. По рекомендациям Oracle, этот эффект начинает проявляться примерно от 15 000 одновременно откомпилированных программ.

Низкоуровневая компиляция и версии Oracle

Способ настройки низкоуровневой компиляции и выполнения зависит от версии базы данных. Подробности приведены в главе 20, а ниже приведена краткая сводка возможностей компиляции в разных версиях.

Oracle9i Database. Возможность низкоуровневой компиляции программ появилась в Oracle9i. В этой версии компиляция работала неплохо... если вы не использовали технологию RAC (Real Application Clusters) и не возражали против сложной процедуры резервного копирования. Базы данных RAC создавали проблемы (они не поддерживались), а резервная копия базы данных должна была включать общие библиотеки, которые не сохранялись программой Oracle Recovery Manager (RMAN).

Oracle Database 10g. В Oracle Database 10g низкоуровневая компиляция была усовершенствована. Базы данных RAC и общие серверы поддерживались, но администратору был необходим компилятор C и копии общих библиотек на каждом узле RAC. Проблемы с резервным копированием оставались — по-прежнему в него необходимо было включать общие библиотеки, но программа RMAN эти библиотеки не сохраняла.

Oracle Database 11g. В Oracle Database 11g низкоуровневая компиляция была снова усовершенствована. Теперь компилятор C перестал быть необходимым, а общие библиотеки хранятся в словаре данных, что позволяет любой программе резервного копирования (по крайней мере, работающей с базами данных Oracle) легко их найти и скопировать. Итак, с исчезновением проблем с резервным копированием и управлением общими библиотечными файлами ничто не мешает вам применять низкоуровневую компиляцию в рабочих и тестовых базах данных. Попробуйте — вам понравится.

Что необходимо знать

Так ли *необходимо* знать все, о чем рассказано в этой главе? Конечно, нет. С другой стороны, ваш администратор базы данных, вероятно, должен знать большую часть этого материала.

Помимо удовлетворения здорового любопытства, этот материал должен был рассеять некоторые распространенные заблуждения по поводу архитектуры PL/SQL. Как бы то ни было, вы должны помнить некоторые важные обстоятельства, относящиеся к внутренним механизмам PL/SQL.

- Чтобы избежать лишних затрат при компиляции, часто вызываемый код следует оформить в виде хранимых программ (вместо анонимных блоков).
- Кроме своей уникальной возможности сохранения состояния во время сеанса, пакеты PL/SQL обладают преимуществами из области производительности. Большая часть логики приложения должна размещаться в телах пакетов.
- При обновлении Oracle новые возможности компилятора PL/SQL необходимо тщательно протестировать. В некоторых (достаточно редких) случаях переход на Oracle11g сопровождается небольшими изменениями в порядке выполнения (в соответствии с решениями, принимаемыми оптимизирующим компилятором), которые могут повлиять на результаты приложения.
- Хотя автоматическое управление зависимостями Oracle избавляет разработчика от многих хлопот, к обновлению приложений на реально используемой базе данных следует относиться в высшей степени осторожно из-за необходимости блокировки объектов и сброса состояния пакетов.
- Если вы используете удаленную проверку зависимостей на базе сигнатур при удаленных вызовах процедур, создайте процедуры для устранения возможности ложноотрицательных срабатываний (приводящих к ошибке времени выполнения).
- Используйте модель разрешений создателя для достижения максимальной производительности, а также упрощения управления и контроля за привилегиями доступа к таблицам баз данных. Модель разрешений вызывающего применяется только для

решения конкретных проблем (например, программ, использующих динамический SQL с созданием или уничтожением объектов баз данных).

- Технологические решения Oracle по минимизации машинных ресурсов, необходимых для выполнения PL/SQL, иногда нуждаются в небольшом содействии со стороны разработчиков и администраторов — например, явном освобождении неиспользуемой памяти.
- Там, где это оправдано логикой приложения, используйте идиому курсорных циклов `FOR` вместо открытия/цикла выборки/закрытия, чтобы пользоваться преимуществами автоматической массовой привязки в Oracle10g и последующих версиях.
- Если ваша программа открывает явный курсор в программе PL/SQL, не забудьте закрыть курсор после завершения выборки данных.
- Низкоуровневая компиляция PL/SQL может не обеспечивать значительного выигрыша в быстродействии приложений, интенсивно использующих SQL, но в программах с большим объемом вычислений выигрыш может быть значительным.
- Используйте программные переменные во встроенных статических командах SQL в PL/SQL и переменные привязки в динамических командах SQL, чтобы не препятствовать совместному использованию курсоров.

25 Глобализация и локализация в PL/SQL

Коммерческие предприятия редко сразу выходят на глобальный уровень. Обычно они начинают с местного или регионального уровня, планируя расширение на будущее. По мере выхода в новые регионы, в которых действуют другие культурные контексты, программное обеспечение приходится адаптировать к новым языкам и требованиям форматирования. Если приложение не спроектировано с расчетом на поддержку нескольких локальных контекстов, такой переход оказывается чрезвычайно длительным и дорогостоящим.

В идеале глобализация является неотъемлемой частью архитектуры приложения, а при принятии всех решений проектировщик задает себе вопрос: «В какой стране это решение не будет работать?» К сожалению, в реальном мире многие компании не включают стратегию глобализации в свою исходную архитектуру. Дополнительные затраты, отсутствие опыта глобализации или просто неспособность предвидеть глобальный характер бизнеса — самые распространенные причины, по которым игнорируются риски локализации. Хотя в чем, собственно, проблема? Разве она не решается простым переводом данных? Нет, не совсем. Многие стандартные задачи программирования PL/SQL имеют последствия, которые могут нарушить работу вашего приложения в другом локальном контексте.

Точность переменных. Переменной CHAR(1) вполне достаточно для хранения символа «F», но хватит ли ее для символа «民»?

Порядок сортировки результата. Порядок ORDER BY легко определяется для английского языка. Будет ли он так же просто определяться в корейском, китайском или японском языке? И что делать с комбинационными символами или символами, содержащими диакритические знаки?

Выборка информации. Язык PL/SQL часто используется для написания информационно-поисковых систем. Как организовать хранение данных на нескольких языках с универсальным поиском по одному запросу?

Формат даты/времени. В разных странах и регионах используются разные календари и форматы данных. Насколько устойчив ваш код к подобным модификациям?

Формат денежных величин. Проблемы выходят за рамки простого преобразования денежных сумм. Неправильно поставленная запятая или точка, соответствующая национальным правилам форматирования денежных величин, может случайно изменить интерпретацию денежной суммы.

В этой главе рассматриваются последствия подобных проблем, а также показано, как пишется код PL/SQL, в котором эти проблемы учитываются и успешно решаются. Мы начнем с обсуждения Юникода и архитектуры Oracle Globalization Support. Далее будут продемонстрированы проблемы, связанные с использованием многобайтовых символов, а также описаны пути их решения в программах PL/SQL. Затем мы обсудим некоторые сложности, связанные с сортировкой символьных строк в разных наборах символов, и изучим эффективную выборку многоязыковой информации. В завершение главы показано, как заставить ваше приложение работать с разными форматами даты/времени и денежных величин.

СТРАТЕГИЯ ГЛОБАЛИЗАЦИИ

Если разработка осуществляется с учетом возможной глобализации, проблемы локализации внезапно проявляются на более ранней стадии жизненного цикла разработки. Об этих аспектах лучше всего думать, находясь еще на фазе проектирования. Конечно, стратегия глобализации слишком сложна, чтобы ее можно было изложить в одной главе, но ваши программы PL/SQL будут находиться по крайней мере в неплохом состоянии, если вы продумаете следующие аспекты:

- Набор символов.
- Параметры NLS.
- Функции Юникода.
- Символьная и байтовая семантика операций.
- Порядок сортировки строк.
- Многоязыковая выборка информации.
- Форматирование даты/времени.
- Форматирование денежных сумм.

За дополнительной информацией о глобализации и локализации в Oracle обращайтесь к разделу «Globalization Support» на сайте Oracle. Здесь вы найдете форумы, ссылки на статьи и дополнительную документацию по глобализации баз данных и серверов приложений.



В этой главе мы будем работать со схемой g11n. Если вы захотите установить этот каталог в своей среде, загрузите файл G11N.ZIP с сайта книги и распакуйте его в свой локальный каталог. Включите в заголовок g11n.sql правильные значения параметров для вашей системы, но будьте внимательны и проследите за тем, чтобы сохранение осуществлялось в Юникоде. Если файлы сохраняются в ASCII или другой западной кодировке, многобайтовые символы в файле будут сохранены некорректно, и при запуске сценариев возникнут ошибки. Сценарий g11n.sql создает пользователя с именем g11n, представляет ему привилегии для работы с примерами, создает объекты и добавляет начальные данные. За дополнительными инструкциями обращайтесь к заголовку файла g11n.sql.

Общие сведения и терминология

Прежде чем переходить к основному материалу, давайте разберемся с терминологией. Термины «глобализация», «интернационализация» и «локализация» часто используются как синонимы, однако они имеют совершенно разный смысл.

В табл. 25.1 объясняется смысл каждого термина и связь между ними.

Таблица 25.1. Терминология 1

| Термин | Общепринятое сокращение | Определение |
|---------------------|-------------------------|--|
| Глобализация | g11n | Стратегия разработки приложения, ориентированная на многоязыковую поддержку и независимость от локального контекста. Глобализация достигается посредством интернационализации и локализации |
| Интернационализация | i18n | Проектирование или модификация приложения, направленные на работу во многих локальных контекстах |
| Локализация | l10n | Процесс внесения изменений, обеспечивающих работу приложения в каждом конкретном локальном контексте. В частности, локализация включает перевод текстов; задача упрощается при качественном выполнении интернационализации |



Если вы еще не видели сокращения из табл. 25.1, обратите внимание на числа между буквами. Сокращение состоит из первой и последней буквы каждого термина, между которыми заключено количество символов. Например, в слове «globalization» между «g» и «n» заключено 11 букв, отсюда и сокращение g11n.

Oracle поддерживает локализацию для любой части света. Впрочем, иногда приходится слышать мнение, что полноценная локализация Oracle означает, что вы можете ввести данные на английском языке и вести поиск в них на японском. Ничего подобного! Oracle не содержит встроенного ядра, которое могло бы на ходу переводить для вас текст. Впрочем, если вы когда-нибудь видели результаты машинного перевода, вряд ли вам захочется пользоваться встроенными «функциями» такого рода. Oracle *поддерживает* локализацию, но не реализует ее за вас. Этим приходится заниматься самому программисту.

Таблица 25.2. Терминология 2

| Термин | Определение |
|-----------------------|--|
| Кодировка | Каждый символ является представлением кодовой точки (code point). Кодировка является отображением между символами и кодовыми точками. Тип кодировки, выбранной для базы данных, определяет возможность хранения и выборки кодовых точек |
| Набор символов | Символы группируются в соответствии с языком или регионом. Множество символов, относящихся к некоторому региону, называется набором символов |
| Кодовая точка | Каждый символ в каждом наборе имеет уникальный идентификатор, называемый кодовой точкой. Этот идентификатор определяется Консорциумом Юникода. Кодовая точка представляет символ в целом или может объединяться с другими кодовыми точками для формирования составных символов. Пример кодовой точки — \0053 |
| Глиф | Глифом называется графическое представление символа, связанное с одной или несколькими кодовыми точками. Глиф, связанный с кодовой точкой \0053, представляет собой прописную букву S |
| Многобайтовые символы | Для представления символов большинства западно-европейских языков хватает одного байта. Многобайтовые символы (японские, корейские и т. д.) используют от 2 до 4 байтов для хранения одного символа в базе данных |
| NLS | NLS (National Language Support) — старое название архитектуры глобализации Oracle. Начиная с Oracle9i используется новый термин Globalization Support, но в документации и параметрах время от времени встречается старое сокращение NLS |
| Юникод | Стандарт кодировки символов |

Знакомство с Юникодом

До разработки стандарта Юникод существовало множество схем кодировки, которые обладали ограниченными возможностями, а порой и конфликтовали друг с другом.

Разработка глобальных приложений по единым правилам была практически невозможна, потому что ни одна кодировка не поддерживала все символы.

Стандарт Юникод решает все эти проблемы. Он разрабатывается и сопровождается Консорциумом Юникода. Содержимое каждой версии определяется Стандартом Юникода и Базой данных символов Юникода, или USD (Unicode Character Database).

Набор символов Юникода позволяет хранить и извлекать данные в более чем 200 различных отдельных наборах. Использование набора символов Юникода обеспечивает поддержку всех этих наборов без внесения архитектурных изменений в приложение.

- Oracle11g Release 2 поддерживает Юникод версии 5.0. Этот стандарт, впервые опубликованный в 2006 году, обеспечивает кодирование более одного миллиона символов. Этого достаточно для поддержки всех современных символов, а также многих древних или малораспространенных алфавитов. Oracle Database 12c включает поддержку Юникода 6.1 (стандарт опубликован в январе 2012 г.) и вводит несколько новых лингвистических порядков сопоставления, соответствующих правилам UCA (Unicode Collation Algorithm).
- Наборы символов Юникода в Oracle11g включают кодировки UTF-8 и UTF-16. В UTF-8 для представления символа используется 1, 2 или 3 байта в зависимости от символа. В UTF-16 символ всегда представляется двумя байтами. В обеих схемах поддерживаются дополнительные символы, использующие 4-байтовое представление независимо от выбранного набора символов Юникода.
- Наборы символов Юникода в Oracle Database 11g и 12c включают кодировки UTF-8 и UTF-16. В UTF-8 символы представляются 1, 2 или 3 байтами в зависимости от символа. В UTF-16 все символы представляются 2 байтами. Дополнительные символы поддерживаются обеими кодировками и представляются 4 байтами на символ независимо от выбранной кодировки.

Каждая база данных Oracle имеет два набора символов. Первичный набор символов используется для большинства функций приложений, а отдельный набор символов NLS — для типов данных и функций, специфических для NLS. Для определения используемых наборов символов используется следующий запрос:

```
SELECT parameter, VALUE
FROM nls_database_parameters
WHERE parameter IN ('NLS_CHARACTERSET', 'NLS_NCHAR_CHARACTERSET')
```

В рабочей среде автора запрос возвращает следующий результат:

| PARAMETER | VALUE |
|------------------------|-----------|
| NLS_CHARACTERSET | AL32UTF8 |
| NLS_NCHAR_CHARACTERSET | AL16UTF16 |

В данном случае параметр NLS_CHARACTERSET (первичный набор символов базы данных) имеет значение AL32UTF8. В этот 32-разрядный набор символов Юникода UTF-8 входит большинство самых распространенных символов в мире. Параметр NLS_NCHAR_CHARACTERSET, используемый прежде всего для столбцов NCHAR и NVARCHAR2, представляет собой 16-разрядный набор символов UTF-16.

Структура имен, присваиваемых наборам символов в Oracle, содержит полезную информацию. Например, US7ASCII поддерживает символы английского языка для США. Набор символов AL32UTF8 поддерживает *любые* языки. Вторая часть строки определяет количество битов на символ. В US7ASCII символ представляется 7 битами, а AL32UTF8 использует до 32 бит на символ. Оставшаяся часть строки содержит «официальное» название набора символов. Структура имени представлена на рис. 25.1.

ВЫБОР НАБОРА СИМВОЛОВ

Во всех новых установках Oracle рекомендуется задавать в параметре `NLS_CHARACTERSET` набор символов Юникода, и это хороший совет. Возможно, прямо сейчас вашему приложению хватает ASCII, но что произойдет через два или три года? Использование Юникода не оказывает сколько-нибудь заметного влияния на быстродействие, а дополнительные затраты места ничтожны, поскольку кодировка использует переменный размер в байтах в зависимости от самих символов.

Даже если вы не планируете работать с данными на разных языках, следует учитывать еще один фактор: в вашей базе данных могут оказаться многобайтовые символы. Браузерные приложения часто поддерживают копирование/вставку больших блоков текста из программ-редакторов. При этом они не ограничиваются простыми ASCII-символами. Скажем, маркеры в маркированных списках представляют собой многобайтовые символы. Без анализа всех данных в текстовых полях вам будет трудно определить, поддерживаются ли входные данные действующим однобайтовым набором символов. Юникод гарантирует, что база данных успешно справится с теми символами, с которыми вы работаете сегодня — и будете работать завтра.

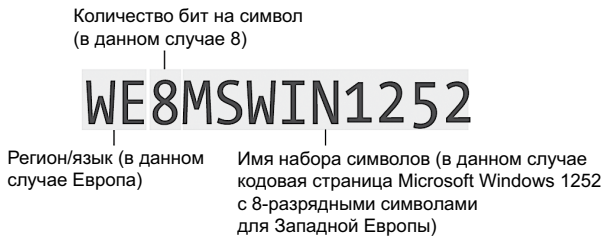


Рис. 25.1. Структура имени набора символов в Oracle

За дополнительной информацией о Юникоде обращайтесь на сайт Стандарта Юникод по адресу <http://unicode.org/unicode/standard/standard.html>.

Типы данных и национальные наборы символов

Типы данных Globalization Support `NCLOB`, `NCHAR` и `NVARCHAR2` используют набор символов, определяемый параметром `NLS_NCHAR_CHARACTERSET`, — вместо набора символов по умолчанию, устанавливаемого для базы данных в параметре `NLS_CHARACTERSET`. Эти типы данных поддерживают только многобайтовые символы Юникода, поэтому даже при работе с базой данных, в которой по умолчанию вместо Юникода используется другая кодировка, они будут хранить символы в национальном наборе символов. А так как национальный набор символов поддерживает только кодировки UTF-8 и UTF-16, `NCLOB`, `NCHAR` и `NVARCHAR2` гарантированно будут хранить данные в многобайтовом Юникоде.

Прежде это создавало проблемы при сравнении столбцов `NCLOB`/`NCHAR`/`NVARCHAR2` со столбцами `CLOB`/`CHAR`/`VARCHAR2`. Во всех версиях, поддерживаемых в настоящее время, Oracle выполняет автоматическое преобразование, благодаря которому становится возможным корректное сравнение.

Кодировка символов

Выбор набора символов во время создания базы данных определяет тип кодировки символов. Каждому символу ставится в соответствие код, уникальный для данного символа (кодовая точка). Это значение является частью таблицы отображения символов Юникода, содержимое которой находится под контролем Консорциума Юникода.

Кодовые точки состоят из префикса U+ (или обратной косой черты \), за которым следует шестнадцатеричный код символа с диапазоном допустимых значений от U+0000 до U+10FFFF¹⁶. Комбинированные символы (например, Ä) могут разбиваться на компоненты (A с умляутом), а затем снова восстанавливаться в своем исходном состоянии. Скажем, декомпозиция Ä состоит из кодовых точек U+0041 (A) и U+0308 (умляут). В следующем разделе будут рассмотрены некоторые функции Oracle для работы с кодовыми точками.

Кодовой единицей (code unit) называется размер в байтах типа данных, используемого для хранения символов. Размер кодовой единицы зависит от используемого набора символов. В некоторых обстоятельствах кодовая точка слишком велика для одной кодовой единицы, и для ее представления требуется несколько кодовых единиц.

Конечно, пользователи воспринимают символы, а не кодовые точки или кодовые единицы. «Слово» \0053\0074\0065\0076\0065\006E вряд ли будет понятно среднему пользователю, который распознает символы на своем родном языке. Не забывайте, что *глиф* (изображение символа, непосредственно отображаемое на экране) является всего лишь представлением кодового пункта. Даже если на вашем компьютере не установлены необходимые шрифты или он по другим причинам не может вывести символы на экран, это вовсе не означает, что в Oracle соответствующая кодовая точка хранится некорректно.

ЮНИКОД И ВАША РАБОЧАЯ СРЕДА

Oracle поддерживает все возможные символы... но поддерживает ли их ваша среда? Если вы не работаете постоянно с символами Юникода, существует большая вероятность того, что ваша система не настроена на поддержку некоторых многобайтовых символов (то есть не сможет воспроизвести правильные глифы). Поддержка Юникода операционной системой не гарантирует, что все приложения в этой операционной системе будут работать со всеми символами. Разработчики приложений сами управляют поддержкой Юникода в своих продуктах. Даже в основных приложениях (и в DOS) при неправильной настройке могут возникнуть сложности с конкретными символами.

Если вы хотите, чтобы многобайтовые символы поддерживались при работе с базой данных Oracle, но не хотите менять настройки операционной системы и приложений, попробуйте воспользоваться Oracle Application Express от Oracle. Вы можете работать с базой данных из браузера, где кодировка Юникод настраивается очень просто. Oracle Application Express бесплатно устанавливается в любой версии базы данных Oracle, и включается в поставку версии Oracle Express Edition. iSQL*Plus — другой вариант для Oracle9i Database и выше.

Многие веб-инструменты также включают соответствующую схему кодировки в заголовки страницы, чтобы символы Юникода правильно отображались по умолчанию. Если же символы отображаются неправильно, задайте кодировку в браузере:

В Internet Explorer выполните команду Вид ► Кодировка ► Автовыбор или Юникод (UTF-8).

В Firefox выполните команду Вид ► Кодировка ► Юникод (UTF-8).

Параметры Globalization Support (NLS)

Поведение Oracle по умолчанию определяется параметрами Globalization Support (NLS). Значения параметров, задаваемые при создании базы данных, определяют многие аспекты ее работы — от наборов символов до используемых по умолчанию денежных единиц. В табл. 25.3 перечислены параметры, которые вы можете изменить в ходе сеанса, с примерами значений и пояснениями. За текущими значениями параметров в вашей системе обращайтесь к представлению NLS_SESSION_PARAMETERS.

Таблица 25.3. Сеансовые параметры NLS

| Параметр | Описание | Пример |
|-------------------------|---|-------------------------------|
| NLS_CALENDAR | Задаёт календарь по умолчанию для базы данных | GREGORIAN |
| NLS_COMP | В сочетании с NLS_SORT определяет правила сортировки символов. При использовании значения ANSI необходимо использовать лингвистический индекс | BINARY |
| NLS_CURRENCY | Задаёт знак денежной единицы; по умолчанию определяется на основании NLS_TERRITORY | \$ |
| NLS_DATE_FORMAT | Задаёт формат даты; по умолчанию определяется на основании NLS_TERRITORY | DD-MON-RR |
| NLS_DATE_LANGUAGE | Определяет способ записи дня и месяца для функций обработки дат | AMERICAN |
| NLS_DUAL_CURRENCY | Упрощает поддержку евро; по умолчанию определяется на основании NLS_TERRITORY. Задаёт альтернативную денежную единицу для территории | \$ |
| NLS_ISO_CURRENCY | Символ денежной единицы ISO; по умолчанию определяется на основании NLS_TERRITORY | AMERICA |
| NLS_LANGUAGE | Задаёт язык, используемый базой данных по умолчанию. Значение влияет на многие аспекты, от формата даты до сообщений сервера | AMERICAN |
| NLS_LENGTH_SEMANTICS | Определяет используемую семантику длины (символы или байты) | BYTE |
| NLS_NCHAR_CONV_EXCP | Определяет, должно ли выдаваться сообщение об ошибке при преобразовании символьного типа | FALSE |
| NLS_NUMERIC_CHARACTERS | Определяет разделители дробной части и групп разрядов; по умолчанию определяется на основании NLS_TERRITORY | „ |
| NLS_SORT | Определяет порядок сортировки символов для заданного языка | BINARY |
| NLS_TERRITORY | Определяет значения по умолчанию многих параметров NLS. Значение определяет основной регион базы данных | AMERICA |
| NLS_TIMESTAMP_FORMAT | Определяет формат временной метки по умолчанию для функций TO_TIMESTAMP и TO_CHAR | DD-MON-RR HH.MI.SSXF F AM |
| NLS_TIMESTAMP_TZ_FORMAT | Определяет формат временной метки с часовым поясом для функций TO_TIMESTAMP и TO_CHAR | DD-MON-RR HH.MI.SSXF F AM TZR |
| NLS_TIME_FORMAT | Используется в сочетании с NLS_DATE_FORMAT (см. выше). Задаёт формат времени по умолчанию для базы данных | HH.MI.SSXF F AM |
| NLS_TIME_TZ_FORMAT | Определяет формат времени с часовым поясом или смещением UTC | HH.MI.SSXF F AM TZR |

Функции Юникода

Поддержка Юникода в PL/SQL начинается с простейших строковых функций. Впрочем, в табл. 25.4 видны небольшие отличия этих функций от их хорошо известных аналогов.

К именам функций `INSTR`, `LENGTH` и `SUBSTR` добавляется суффикс `B`, `C`, `2` или `4`; он означает, что функция работает с байтами, символами, кодовыми единицами или кодовыми точками соответственно.



Функции `INSTR`, `LENGTH` и `SUBSTR` используют семантику длины, связанную с типом данных столбца или переменной. Эти базовые функции и версии с суффиксом `C` часто возвращают одинаковые значения — до тех пор, пока вы не начнете работать со значениями `NCHAR` или `NVARCHAR`. Поскольку `NLS_NCHAR_CHARACTERSET` и `NLS_CHARACTERSET` могут различаться, результат вызова `INSTR`, `LENGTH` и `SUBSTR` может отличаться (в зависимости от типа данных) от результата их символьных аналогов.

Таблица 25.4. Функции Юникода

| Функция Юникода | Описание |
|---|---|
| <code>ASCIISTR(string)</code> | Преобразует строку <code>string</code> в ASCII-символы. Строки в Юникоде преобразуются в стандартный формат <code>\xxxx</code> |
| <code>COMPOSE(string)</code> | Преобразует строку <code>string</code> , полученную в результате декомпозиции, в полную композиционную форму |
| <code>DECOMPOSE(string, [CANONICAL COMPATIBILITY])</code> | Получает строку <code>string</code> и возвращает строку Юникода, полученную разложением составных символов на кодовые точки |
| <code>INSTRB(string, substr, pos, occ)</code> | Возвращает позицию подстроки <code>substr</code> в строке <code>string</code> в байтах, начиная с позиции <code>pos</code> . Аргумент <code>occ</code> задает номер вхождения <code>substr</code> , если подстрока встречается более одного раза. По умолчанию аргументы <code>pos</code> и <code>occ</code> равны 1. Значение <code>pos</code> задается в байтах |
| <code>INSTRC(string, substr, pos, occ)</code> | Аналог <code>INSTRB</code> — за исключением того, что возвращает позицию <code>substr</code> в <code>string</code> в символах, начиная с позиции <code>pos</code> (значение <code>pos</code> задается в символах) |
| <code>INSTR2(string, substr, pos, occ)</code> | Возвращаемая позиция задается в кодовых единицах UTF-16 |
| <code>INSTR4(string, substr, pos, occ)</code> | Возвращаемая позиция задается в кодовых точках UTF-16 |
| <code>LENGTHB(string)</code> | Возвращает размер строки <code>string</code> в байтах |
| <code>LENGTHC(string)</code> | Возвращает длину строки <code>string</code> в символах Юникода |
| <code>LENGTH2(string)</code> | Возвращаемая длина задается в кодовых единицах UTF-16 |
| <code>LENGTH4(string)</code> | Возвращаемая длина задается в кодовых точках UTF-16 |
| <code>SUBSTRB(string, n, m)</code> | Возвращает часть строки <code>string</code> , состоящую из <code>m</code> символов, начиная с позиции <code>n</code> . Значения <code>n</code> и <code>m</code> задаются в байтах |
| <code>SUBSTRC(string, n, m)</code> | Возвращает часть строки <code>string</code> , состоящую из <code>m</code> символов, начиная с позиции <code>n</code> . Значения <code>n</code> и <code>m</code> задаются в символах Юникода |
| <code>SUBSTR2(string, n, m)</code> | Значения <code>n</code> и <code>m</code> задаются в кодовых единицах UTF-16 |
| <code>SUBSTR4(string, n, m)</code> | Значения <code>n</code> и <code>m</code> задаются в кодовых точках UTF-16 |
| <code>UNISTR(string)</code> | Преобразует представление строки <code>string</code> из ASCII-формата (обратная косая черта, шестнадцатеричные цифры) в Юникод |

Рассмотрим эти функции подробнее.

ASCIISTR

`ASCIISTR` пытается преобразовать полученную строку в ASCII-символы. Если строка содержит символы, отсутствующие в наборе ASCII, они представляются в формате `\xxxx`. Как будет показано ниже при описании функции `DECOMPOSE`, такое форматирование иногда оказывается очень удобным.

BEGIN

```
DBMS_OUTPUT.put_line ('ASCII Character: ' || ASCIISTR ('А'));
```

```
DBMS_OUTPUT.put_line ('Unicode Character: ' || ASCIISTR ('А'));
```

END;

Результат:

ASCII Character: A

Unicode Character: \00C4

COMPOSE

Некоторые символы могут иметь несколько вариантов представления кодовых пунктов. Это создает проблемы при сравнении двух значений. Символ Ä может быть представлен как одним кодовым пунктом U+00C4, так и двумя кодовыми пунктами U+0041 (буква A) и U+0308. При сравнении PL/SQL считает, что эти два варианта представления не равны.

```
DECLARE
  v_precomposed  VARCHAR2 (20) := UNISTR ('\00C4');
  v_decomposed   VARCHAR2 (20) := UNISTR ('A\0308');
BEGIN
  IF v_precomposed = v_decomposed
  THEN
    DBMS_OUTPUT.put_line ('==EQUAL==');
  ELSE
    DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
  END IF;
END;
```

Результат:

<>NOT EQUAL<>

Однако после использования функции COMPOSE эти две версии равны:

```
DECLARE
  v_precomposed  VARCHAR2 (20) := UNISTR ('\00C4');
  v_decomposed   VARCHAR2 (20) := COMPOSE (UNISTR ('A\0308'));
BEGIN
  IF v_precomposed = v_decomposed
  THEN
    DBMS_OUTPUT.put_line ('==EQUAL==');
  ELSE
    DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
  END IF;
END;
```

На этот раз сравнение дает другой результат:

==EQUAL==

DECOMPOSE

Как нетрудно догадаться, функция DECOMPOSE является обратной по отношению к COMPOSE: она разбивает составные символы на отдельные кодовые точки или элементы:

```
DECLARE
  v_precomposed  VARCHAR2 (20) := ASCIISTR (DECOMPOSE ('Ä'));
  v_decomposed   VARCHAR2 (20) := 'A\0308';
BEGIN
  IF v_precomposed = v_decomposed
  THEN
    DBMS_OUTPUT.put_line ('==EQUAL==');
  ELSE
    DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
  END IF;
END;
```

Результат:

==EQUAL==

INSTR/INSTRB/INSTRC/INSTR2/INSTR4

Все функции INSTR возвращают позицию подстроки внутри строки и различаются лишь по способу определения позиции. Для демонстрации мы воспользуемся таблицей `publication` из схемы `g11n`.

```
DECLARE
  v_instr      NUMBER (2);
  v_instrb     NUMBER (2);
  v_instrc     NUMBER (2);
  v_instr2     NUMBER (2);
  v_instr4     NUMBER (2);
BEGIN
  SELECT INSTR (title, 'З'),
         INSTRB (title, 'З'),
         INSTRC (title, 'З'),
         INSTR2 (title, 'З'),
         INSTR4 (title, 'З')
  INTO v_instr, v_instrb, v_instrc,
       v_instr2, v_instr4
  FROM publication
  WHERE publication_id = 2;
  DBMS_OUTPUT.put_line ('INSTR of З: ' || v_instr);
  DBMS_OUTPUT.put_line ('INSTRB of З: ' || v_instrb);
  DBMS_OUTPUT.put_line ('INSTRC of З: ' || v_instrc);
  DBMS_OUTPUT.put_line ('INSTR2 of З: ' || v_instr2);
  DBMS_OUTPUT.put_line ('INSTR4 of З: ' || v_instr4);
END;
```

Результат:

```
INSTR of З: 16
INSTRB of З: 20
INSTRC of З: 16
INSTR2 of З: 16
INSTR4 of З: 16
```

Позиция символа `З` отличается только для INSTRB. Одна из полезных особенностей INSTR2 и INSTR4 заключается в том, что они могут использоваться для поиска кодовых точек, не представляющих полные символы. Возвращаясь к примеру с символом `Ä`, упомянут можно включить как подстроку для выполнения поиска.

LENGTH/LENGTHB/LENGTHC/LENGTH2/LENGTH4

Функции LENGTH возвращают длину строки в разных единицах:

LENGTH — возвращает длину строки в символах;

LENGTHB — возвращает длину строки в байтах;

LENGTHC — возвращает длину строки в символах Юникода;

LENGTH2 — возвращает количество кодовых единиц в строке;

LENGTH4 — возвращает количество кодовых точек в строке.

Если строка состоит из композиционных символов, функция LENGTH эквивалентна LENGTHC.

```
DECLARE
  v_length     NUMBER (2);
  v_lengthb    NUMBER (2);
  v_lengthc    NUMBER (2);
  v_length2    NUMBER (2);
  v_length4    NUMBER (2);
```

```

BEGIN
  SELECT LENGTH (title), LENGTHB (title), lengthc (title), length2 (title),
         length4 (title)
    INTO v_length, v_lengthb, v_lengthc, v_length2,
         v_length4
  FROM publication
 WHERE publication_id = 2;
  DBMS_OUTPUT.put_line ('LENGTH of string: ' || v_length);
  DBMS_OUTPUT.put_line ('LENGTHB of string: ' || v_lengthb);
  DBMS_OUTPUT.put_line ('LENGTHC of string: ' || v_lengthc);
  DBMS_OUTPUT.put_line ('LENGTH2 of string: ' || v_length2);
  DBMS_OUTPUT.put_line ('LENGTH4 of string: ' || v_length4);
END;

```

Результат:

```

LENGTH of string: 28
LENGTHB of string: 52
LENGTHC of string: 28
LENGTH2 of string: 28
LENGTH4 of string: 28

```

В данном примере только функция `LENGTHB` дает другой результат. Как и ожидалось, `LENGTH` и `LENGTHC` вернули одинаковые результаты. Впрочем, при работе с декомпозиционными символами ситуация меняется. Пример:

```

DECLARE
  v_length  NUMBER (2);
BEGIN
  SELECT LENGTH (UNISTR ('A\0308'))
    INTO v_length
  FROM DUAL;

  DBMS_OUTPUT.put_line ('Decomposed string size using LENGTH: ' || v_length);
  SELECT lengthc (UNISTR ('A\0308'))
    INTO v_length
  FROM DUAL;

  DBMS_OUTPUT.put_line ('Decomposed string size using LENGTHC: ' || v_length);
END;

```

Функции возвращают следующие значения длины:

```

Decomposed string size using LENGTH: 2
Decomposed string size using LENGTHC: 1

```

Функция `LENGTH` возвращает количество символов, но считает `A` и умляут разными символами. `LENGTHC` возвращает длину в символах Юникода и видит только один символ.

SUBSTR/SUBSTRB/SUBSTRC/SUBSTR2/SUBSTR4

Разные версии `SUBSTR` определяются по тому же принципу, что и их аналоги у функций `INSTR` с `LENGTH`. `SUBSTR` возвращает часть строки заданной длины начиная с заданной позиции. Функции этого семейства работают следующим образом:

```

SUBSTR — определяет позицию и длину по символу;
SUBSTRB — определяет позицию и длину в байтах;
SUBSTRC — определяет позицию и длину в символах Юникода;
SUBSTR2 — использует кодовые единицы;
SUBSTR4 — использует кодовые точки.

```

Использование этих функций продемонстрировано в следующем примере:

```

DECLARE
  v_substr  VARCHAR2 (20);
  v_substrb VARCHAR2 (20);

```

```

v_substrc   VARCHAR2 (20);
v_substr2   VARCHAR2 (20);
v_substr4   VARCHAR2 (20);
BEGIN
  SELECT SUBSTR (title, 13, 4), SUBSTRB (title, 13, 4),
         substrc (title, 13, 4), substr2 (title, 13, 4),
         substr4 (title, 13, 4)
  INTO v_substr, v_substrb,
       v_substrc, v_substr2,
       v_substr4
  FROM publication
  WHERE publication_id = 2;

  DBMS_OUTPUT.put_line ('SUBSTR of string: ' || v_substr);
  DBMS_OUTPUT.put_line ('SUBSTRB of string: ' || v_substrb);
  DBMS_OUTPUT.put_line ('SUBSTRC of string: ' || v_substrc);
  DBMS_OUTPUT.put_line ('SUBSTR2 of string: ' || v_substr2);
  DBMS_OUTPUT.put_line ('SUBSTR4 of string: ' || v_substr4);
END;
```

Обратите внимание на отличие SUBSTRB от других функций в результатах выполнения сценария:

```

SUBSTR of string: Lプログ
SUBSTRB of string: Lﾌ
SUBSTRC of string: Lプログ
SUBSTR2 of string: Lプログ
SUBSTR4 of string: Lプログ
```

UNISTR

Функция UNISTR преобразует строку в Юникод. Эта функция использовалась в ряде предыдущих примеров для вывода символов строки, подвергнутой декомпозиции. В разделе «Кодировка символов» в качестве примера была приведена строка, состоящая из кодовых пунктов. Чтобы привести ее к понятному виду, можно воспользоваться функцией UNISTR:

```

DECLARE
  v_string   VARCHAR2 (20);
BEGIN
  SELECT UNISTR ('\0053\0074\0065\0076\0065\006E')
  INTO v_string
  FROM DUAL;
  DBMS_OUTPUT.put_line (v_string);
END;
```

Результат:

Steven

Символьная семантика

Несомненно, в числе первых проблем, с которыми вы столкнетесь при локализации своих приложений, будет поддержка многобайтовых символов. Когда вы передадите первый японский символ в переменную VARCHAR2 и получите ошибку ORA-6502, скорее всего, час-другой будет потрачен на отладку процедуры, которая «должна работать».

Возможно, в какой-то момент выяснится, что для поддержки многобайтовых наборов символов вам придется изменить все объявления всех символьных переменных или символьных столбцов в вашем приложении. Не отчаивайтесь! После решения всех начальных проблем значительно упростится управление реализацией приложения в будущем. Рассмотрим пример:

```

DECLARE
  v_title  VARCHAR2 (30);
BEGIN
  SELECT title
    INTO v_title
    FROM publication
   WHERE publication_id = 2;
  DBMS_OUTPUT.put_line (v_title);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

Программа выдает следующее исключение:

ORA-06502: PL/SQL: numeric or value error: character string buffer too small

Проблема в том, что точность переменной составляет 30 байтов, а не символов. Во многих азиатских наборах представление одного символа занимает до 3 байтов, поэтому в переменную длины 2 не поместится ни один символ выбранного набора!

При помощи функции LENGTHB мы можем определить фактический размер строки:

```

DECLARE
  v_length_in_bytes  NUMBER (2);
BEGIN
  SELECT LENGTHB (title)
    INTO v_length_in_bytes
    FROM publication
   WHERE publication_id = 2;
  DBMS_OUTPUT.put_line ('String size in bytes: ' || v_length_in_bytes);
END;
```

Результат:

String size in bytes: 52

До выхода Oracle9i наши возможности были ограничены. Самым распространенным решением в Oracle8i было простое умножение максимального количества символов на 3:

```

DECLARE
  v_title  VARCHAR2 (90);
BEGIN
  SELECT title
    INTO v_title
    FROM publication
   WHERE publication_id = 2;

  DBMS_OUTPUT.put_line (v_title);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

Если в вашей системе необходимые глифы могут быть выведены на экран, возвращается следующий результат:

Oracle PL/SQLプログラミング 基礎編 第3版

Обходное решение работает, но выглядит весьма неуклюже. Байтовая семантика и умножение на 3 приводит к нежелательным последствиям для вашего приложения:

- Многие фирмы-разработчики СУБД по умолчанию используют символьную семантику вместо байтовой, что затруднит возможное портирование приложения.
- Если символы занимают не все 3 байта, в переменной или столбце может быть сохранено больше символов, чем предполагалось.

- Автоматическое дополнение типов данных CHAR в Oracle означает, что все 90 байт будут зарезервированы в памяти — независимо от того, используются они или нет. Символьная семантика впервые появилась в Oracle9i. При объявлении переменной размер может задаваться как в байтах, так и в символах. Следующий пример почти полностью совпадает с неудачным примером, приводившимся ранее, — с одним исключением. Взгляните на объявление переменной, чтобы понять, как активизируется символьная семантика:

```
DECLARE
  v_title  VARCHAR2 (30 CHAR);
BEGIN
  SELECT title
    INTO v_title
    FROM publication
   WHERE publication_id = 2;

  DBMS_OUTPUT.put_line (v_title);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

На этот раз программа выводит полную строку:

Oracle PL/SQLプログラミング 基礎編 第3版

Впрочем, и этот метод требует модификации каждой символьной переменной или объявления столбца в вашем приложении. Проблема проще решается переходом от байтовой семантики к символьной на уровне всей базы данных. Для этого достаточно присвоить переменной NLS_LENGTH_SEMANTICS значение CHAR. Текущее значение параметра можно узнать при помощи следующего запроса:

```
SELECT parameter, VALUE
  FROM nls_session_parameters
 WHERE parameter = 'NLS_LENGTH_SEMANTICS'
```

Результат выполнения:

| PARAMETER | VALUE |
|----------------------|-------|
| NLS_LENGTH_SEMANTICS | BYTE |

Также информацию можно получить из представления V\$PARAMETER:

```
SELECT NAME, VALUE
  FROM v$parameter
 WHERE NAME = 'nls_length_semantics'
```

Запрос возвращает следующую информацию:

| NAME | VALUE |
|----------------------|-------|
| nls_length_semantics | BYTE |

Для изменения значения NLS_LENGTH_SEMANTICS используется команда ALTER SYSTEM:

```
ALTER SYSTEM SET NLS_LENGTH_SEMANTICS = CHAR
```

Команда ALTER SESSION изменяет значение параметра на время текущего сеанса:

```
ALTER SESSION SET NLS_LENGTH_SEMANTICS = CHAR
```

При таком подходе модификация существующего приложения выполняется проще простого; все существующие объявления автоматически интерпретируются в символах вместо байтов. После перевода системы на символьную семантику изменения отражаются в словаре данных:

```
SELECT parameter, value
  FROM nls_session_parameters
 WHERE parameter = 'NLS_LENGTH_SEMANTICS'
```

Результат:

| PARAMETER | VALUE |
|----------------------|-------|
| NLS_LENGTH_SEMANTICS | CHAR |

Возвращаясь к предыдущему примеру, мы видим, что символьная семантика применяется без включения ключевого слова CHAR в объявление.

```
DECLARE
  v_title  VARCHAR2 (30);
BEGIN
  SELECT title
    INTO v_title
   FROM publication
   WHERE publication_id = 2;

  DBMS_OUTPUT.put_line (v_title);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

Результат будет таким:

Oracle PL/SQLプログラミング 基礎編 第3版

Обратите внимание: максимальный размер в байтах не изменяется при активизации символьной семантики. Хотя символьная семантика позволит разместить 1000 3-байтовых символов в переменной VARCHAR2(1000) без изменений, поместить 32,767 3-байтовых символов в переменной VARCHAR2(32767) вам не удастся.

Предельный размер переменной VARCHAR2 по-прежнему составляет 32 767 байт, а столбца VARCHAR2 — 4000 байт.



Учитывая символьную семантику при проектировании приложения, вы сильно упростите себе жизнь. Если только у вас нет веских причин для использования байтовой семантики в части вашего приложения, задайте параметр NLS_LENGTH_SEMANTICS = CHAR, чтобы символьная семантика использовалась по умолчанию. Если вы измените настройку NLS_LENGTH_SEMANTICS для существующего приложения, не забудьте перекомпилировать все объекты, чтобы изменения вступили в силу. В частности, вам придется заново выполнить сценарий catproc.sql, чтобы повторно создать все пакеты!

Порядок сортировки строк

Расширенные возможности сортировки Oracle выходят далеко за рамки простейшей сортировки A-Z, которая реализуется секцией ORDER BY. Сложности, встречающиеся в международных наборах символов, не решаются простой алфавитной сортировкой. Скажем, в китайском языке существует около 70 000 символов (хотя не все они встречаются при повседневном использовании). Такое разнообразие явно не укладывается в простую схему сортировки.

О порядке сортировки строк часто забывают в ходе глобализации, пока продукт не доберется до группы тестирования. Упорядочение имен работников, городов или клиентов — задача намного более сложная, чем простые формулировки «А предшествует В». Необходимо учесть следующие факторы:

- В некоторых европейских символах встречаются диакритические элементы, изменяющие смысл базовой буквы. Скажем, буква «а» отличается от «ä». Какая из них должна стоять на первом месте в порядке **ORDER BY**?
- Каждый локальный контекст может иметь собственные правила сортировки, поэтому многоязыковое приложение должно поддерживать разные правила сортировки в зависимости от текста. Даже регионы с одинаковыми алфавитами могут иметь разные правила сортировки.

Oracle поддерживает три вида сортировки: двоичную, одноязычную и многоязычную.

Консорциум Юникода публикует свой алгоритм сортировки, так что мы можем сравнить вывод наших запросов для этих трех типов сортировки с ожидаемыми результатами, приведенными на сайте Юникода.

Двоичная сортировка

Двоичная сортировка основана на кодировке символов. Она работает очень быстро и особенно удобна при работе с данными, которые хранятся в верхнем регистре. Двоичная сортировка чаще всего применяется для ASCII-текста и английского алфавита, но даже в этом случае возможны некоторые нежелательные результаты. Скажем, в ASCII буквы верхнего регистра располагаются до их представлений в нижнем регистре. Следующий пример из схемы g11n демонстрирует результаты двоичной сортировки названий городов в Германии:

```
SELECT city
  FROM store_location
 WHERE country <> 'JP'
ORDER BY city;
```

Отсортированный список результатов выглядит так:

```
CITY
-----
Abdëra
Asselfingen
Astert
Auufer
Außernzell
Aßlar
Boßdorf
Bösleben
Bötersen
Cremlingen
Creuzburg
Creußen
Oberahr
Zudar
Zühlen
Ängelholm
...lsen
```

Обратите внимание на порядок следования городов в списке: Ängelholm следует после Zühlen. Коды символов сортируются по возрастанию; так формируется порядок A–Z в приведенном листинге. Аномалии возникают из-за символов, отсутствующих в английском алфавите.

Одноязычная сортировка

Средства одноязычной сортировки Oracle пригодятся при работе со многими европейскими языками. Вместо базовых кодов в схеме кодировки символов, как при двоичной сортировке, позиция символа при одноязычной сортировке определяется двумя

значениями. С каждым символом связывается основное значение, соответствующее базовому символу, и дополнительное значение, определяемое регистром и различиями в диакритических элементах. При несовпадении основных значений порядок сортировки определяется однозначно. Если основные значения совпадают, используется дополнительное значение. Таким образом обеспечивается правильный порядок следования символа «ö» по отношению к «o».

Чтобы увидеть, как выбор этого типа сортировки влияет на упорядочение «нестандартных» символов, вернемся к предыдущему примеру и включим для текущего сеанса одноязычную сортировку для немецкого языка:

```
ALTER SESSION SET NLS_SORT = german;
```

Получив подтверждение об изменении настроек сеанса, выполним следующий запрос:

```
SELECT city
       FROM store_location
      WHERE country <> 'JP'
 ORDER BY city;
```

Обратите внимание на изменение порядка названий городов:

```
CITY
-----
Abdêra
Ängelholm
Aßlar
Asselfingen
Astert
Außernzell
Auufer
Boßdorf
Bösleben
Bötersen
Cremlingen
Creußen
Creuzburg
Oberahr
...lsen
Zudar
Zühlen
```

Гораздо лучше! Порядок следования символов, не входящих в английский алфавит, теперь соответствует правилам немецкого языка. Кстати говоря, если вы не хотите (или не можете) изменять сеансовые настройки NLS, используйте функцию NLSSORT и параметр NLS_SORT в составе запроса:

```
FUNCTION city_order_by_func (v_order_by IN VARCHAR2)
    RETURN sys_refcursor
IS
    v_city sys_refcursor;
BEGIN
    OPEN v_city
    FOR
        SELECT city
           FROM store_location
          ORDER BY NLSSORT (city, 'NLS_SORT=' || v_order_by);

    RETURN v_city;
END city_order_by_func;
```

Функция NLSSORT и параметр NLS_SORT предоставляют простые средства для изменения результатов ORDER BY. Приведенная функция, используемая в последующих примерах, получает параметр NLS_SORT во входных данных. В табл. 25.5 перечислены некоторые значения параметра NLS_SORT, доступные в Oracle11g.

Таблица 25.5. Значения параметра NLS_SORT для одноязычной сортировки

| | | |
|------------------|-------------|----------------|
| arabic | xcatalan | japanese |
| arabic_match | german | polish |
| arabic_abj_sort | xgerman | punctuation |
| arabic_abj_match | german_din | xpunctuation |
| azerbaijani | xgerman_din | romanian |
| xazerbaijani | hungarian | russian |
| bengali | xhungarian | spanish |
| bulgarian | icelandic | xspanish |
| canadian french | indonesian | west_european |
| catalan | italian | xwest_european |

Некоторые значения в этом списке начинаются с префикса x: это расширенные режимы сортировки для особых случаев в языке. В приведенном примере с городами некоторые названия содержат символ ß. В немецком языке при сортировке этот символ может интерпретироваться как последовательность «ss». Ранее мы использовали сортировку со значением NLS_SORT = `german`. Давайте посмотрим, какой результат будет получен в режиме `xgerman`:

```
VARIABLE v_city_order REFCURSOR
CALL city_order_by_func('xgerman') INTO :v_city_order;
PRINT v_city_order
```

Результат:

```
CITY
-----
...
Abdêra
Ängelholm
Asselfingen
Aßlar
Astert
Außernzell
Auufer
...
```

В режиме `xgerman` слово `Aßlar` переходит с третьего места в списке на четвертое.

Многоязычная сортировка

Как нетрудно догадаться, одноязычная сортировка обладает серьезным недостатком: она работает только с одним языком, заданным параметром NLS_SORT. Oracle также предоставляет многоязычные средства сортировки, позволяющие работать с несколькими локальными контекстами.

Многоязычная сортировка, базирующаяся на стандарте ISO 14651, поддерживает более 1,1 миллиона символов. Oracle поддерживает не только символы, определяемые в стандарте Юникода 4.0, но и некоторые дополнительные символы.

В отличие от двухэтапной одноязычной сортировки, многоязычная сортировка определяет порядок символов в три этапа:

1. На первом уровне отделяются базовые символы.
2. На втором уровне базовые символы отделяются от диакритических элементов, модифицирующих базовые символы.
3. На третьем уровне происходит разделение символов по регистру.

Функция NLSSORT и параметр NLS_SORT используются и при многоязычной сортировке, но при этом используются другие значения. Режим `GENERIC_M` хорошо работает

в большинстве западных языков. В табл. 25.6 перечислены значения параметра NLS_SORT, доступные для многоязычной сортировки.

Таблица 25.6. Значения параметра NLS_SORT для многоязычной сортировки

| | | | |
|------------|-------------------|--------------------|--------------------|
| generic_m | | | |
| canadian_m | japanese_m | schinese_pinyin_m | tchinese_radical_m |
| danish_m | korean_m | schinese_radical_m | tchinese_stroke_m |
| french_m | schinese_stroke_m | spanish_m | thai_m |

Чтобы продемонстрировать режим многоязычной сортировки, мы изменим вызов функции так, чтобы в нем использовалось значение `generic_m`:

```
VARIABLE v_city_order REFCURSOR
CALL city_order_by_func('generic_m') INTO :v_city_order;
PRINT v_city_order
```

Упорядоченный список городов выглядит так:

```
CITY
-----
Abdëra
Ängelholm
Asselfingen
Aßlar
Astert
..
Zudar
Zühlen
尼崎市
旭川市
足立区
青森市
```

Многоязыковой информационный поиск

Разработчикам приложений, работающим с каталогами, цифровыми библиотеками и репозиториями, часто решают задачу поиска записей или документов, ближе всего соответствующих заданному критерию и *намерениям* пользователя. Это одно из принципиальных отличий получения информации от стандартных запросов SQL, которые либо находят совпадения для критерия запроса, либо не находят их. Хорошая система информационного поиска может определить содержимое документа и вернуть документы, в наибольшей степени относящиеся к критерию поиска, даже при отсутствии точного совпадения.

Вероятно, самой сложной задачей в области информационного поиска является поддержка индексирования и запросов на разных языках. Например, однобайтовый английский язык использует пробелы для разделения слов. При работе с японским языком, использующим многобайтовый набор символов *без* пробелов-разделителей, информационный поиск производится совершенно иначе.

Oracle Text — программа, входящая и в Oracle Enterprise Edition, и в Oracle Standard Edition — обеспечивает функциональность полнотекстового информационного поиска. Благодаря использованию SQL для создания индексов, поиска и операций сопровождений, Oracle Text очень хорошо работает в сочетании с приложениями на базе PL/SQL. Программа Oracle Text (в предыдущих версиях называвшаяся ConText and interMedia) стала полноценным решением в области информационного поиска с выходом Oracle9i. Oracle Text:

- Поддерживает все наборы символов NLS.
- Делает возможным поиск по документам на западных языках, а также на корейском, японском, традиционном и упрощенном китайском.
- Учитывает уникальные характеристики всех языков.
- По умолчанию выполняет поиск без учета регистра символов.
- Поддерживает межязыковой поиск.

Прежде чем писать приложение PL/SQL для поиска в источнике данных, необходимо создать индексы Oracle Text. Я создал индекс Oracle Text по столбцу `publication.short_description` в составе схемы `g11n`. Чтобы обеспечить поддержку нескольких языков, я определил настройки для разных языков, а также настройку `MULTI_LEXER`, позволяющую проводить поиск по нескольким языкам в одном запросе.

ИНДЕКСЫ ORACLE TEXT

В Oracle Text доступны индексы четырех типов. К первому (и самому распространенному) относятся индексы `CONTEXT`. Индекс `CONTEXT` позволяет индексировать любые столбцы символьного типа или типа `LOB`, включая `BFILE`. Он используется для извлечения текста из документов разных типов. Фильтр, поставляемый с сервером данных, может фильтровать более 350 разных типов документов, включая документы Word, PDF и XML.

Текст, извлеченный из документа, разбивается на лексемы (отдельные конструкции и фразы) лексическим анализатором (`LEXER`). При необходимости можно найти лексические анализаторы для разных языков. Объект `MULTI_LEXER` в действительности использует лексические анализаторы отдельных языков (определяемые как `SUB_LEXER`) для извлечения лексем из многоязыкового источника данных. Лексемы хранятся в индексных таблицах Oracle Text и используются во время операций поиска для ссылок на актуальные документы. Чтобы увидеть лексемы, созданные в примерах этой главы, выполните следующий запрос:

```
SELECT token_text
FROM dr$g11n_index$i
```

Результат содержит лексемы для английского, немецкого и японского языков.

Три других типа индексов Oracle Text — `CTXCAT`, `CTXRULE` и `CTXPATH`. За дополнительной информацией об их структуре обращайтесь к документации Oracle Text Application Developer's Guide and Text Reference.

Для тестирования некоторых многоязыковых средств можно воспользоваться функцией `TEXT_SEARCH_FUNC`, являющейся частью схемы `g11n`:

```
FUNCTION text_search_func (v_keyword IN VARCHAR2)
RETURN sys_refcursor
IS
  v_title sys_refcursor;
BEGIN
  OPEN v_title
  FOR
    SELECT title, LANGUAGE, score (1)
    FROM publication
    WHERE contains (short_description, v_keyword, 1) > 0
    ORDER BY score (1) DESC;
  RETURN v_title;
END text_search_func;
```

Вызов этой функции с передачей ключевого слова «pl»:

```
variable x refcursor;
call text_search_func('pl') into :x;
print x;
```

возвращает следующий результат:

| TITLE | LANGUAGE | SCORE(1) |
|--|----------|----------|
| Oracle PL/SQLプログラミング 基礎編 第3版 | JA | 18 |
| Oracle PL/SQL Programming, 3rd Edition | EN | 13 |
| Oracle PL/SQL Programmierung, 2. Auflage | DE | 9 |

Книга находится на всех трех языках, потому что «pl» присутствует во всех вариантах ее названия. Обратите внимание: я провожу поиск по строке «pl» в нижнем регистре, но в записи символы «PL» хранятся в верхнем регистре. По умолчанию поиск проводится без учета регистра, несмотря на использование функции UPPER.

Может оказаться, что в одних языках регистр символов должен игнорироваться, а в других — нет. Признак игнорирования регистра можно задать на уровне отдельных языков в языковых настройках. Просто добавьте атрибут `mixed_case` со значением `yes`; лексемы будут создаваться в том виде, в котором они хранятся в документе или столбце, — но только для языка, указанного в настройке.

В Oracle Database 11g многоязыковой информационный поиск упростился с появлением реализации `AUTO_LEXER`. По количеству функций уровня отдельных языков она уступает `MULTI_LEXER`, но зато почти не требует лишних усилий со стороны разработчика. Вместо того чтобы полагаться на содержимое столбца `language`, `AUTO_LEXER` идентифицирует текст на основании кодовых точек.

Oracle также поддерживает реализацию `WORLD_LEXER`. По широте функций она уступает `MULTI_LEXER` и, как и `AUTO_LEXER`, не предусматривает настройки на уровне отдельных языков. С другой стороны, она очень проста в настройке.

При использовании `WORLD_LEXER` текст разбивается на лексемы в зависимости от категории, к которой он относится. Разбивка текстов на языках арабских и латинских категорий, в которых лексемы разделяются пробелами, не создает проблем. С азиатскими символами дело обстоит сложнее, потому что в них разделители-пробелы не используются, поэтому разбивка осуществляется с перекрытием. Например, трехсимвольная строка 尼崎市 разбивается на две лексемы, 尼崎 и 崎市.

Oracle Text также предоставляет дополнительные возможности в зависимости от языка. За дополнительной информацией, включая возможности и ограничения конкретных языков, обращайтесь к документации Oracle Text на сайте OTN.

Информационный поиск и PL/SQL

Мне доводилось проектировать и реализовывать исключительно большие и сложные системы управления записями и цифровые библиотеки. По собственному опыту могу сказать, что ничто не сравнится с PL/SQL для операций поиска и сопровождения в Oracle Text. Тесная интеграция PL/SQL с сервером базы данных, а также улучшение его быстроедействия в последних версиях делают хранимые программы PL/SQL идеальным вариантом для программ такого рода.

Преимущества становятся еще более очевидными при работе на нескольких языках. Общий разбор SQL и PL/SQL означает логически последовательную обработку символов и символьной семантики независимо от языка, на котором производятся индексирование и поиск.


```

        v_final_search_string := v_token_array (y);
    END IF;
END LOOP;

-- Экранирование специальных символов в строке
v_final_search_string :=
    TRIM (REPLACE (REPLACE (v_final_search_string,
        '&',
        '&',
        ';',
        ';',
        ')
    );
RETURN (v_final_search_string);
END format_string;
```

Эта программа выделяет лексемы из строки по пробелам между символами. Она использует символьную семантику объявления переменных, включая объявление ассоциативного массива.

Чтобы протестировать ее со строкой на английском языке, я выполняю следующую команду SELECT:

```
SELECT format_string('oracle PL/SQL') AS "Formatted String"
FROM dual
```

Команда возвращает следующий результат:

```
Formatted String
-----
ORACLE, PL/SQL
```

Функция `FORMAT_STRING` по умолчанию разделяет лексемы запятыми, так что точное совпадение не обязательно. Строка символов, не ограниченная пробелами, ищется в том виде, в котором она была введена. Следующий пример демонстрирует смешанное использование английских и японских символов:

```
SELECT format_string('Oracle PL/SQLプログラミング 基礎編 第3版') AS
"Formatted String" FROM dual;
```

При передаче этой смешанной строки функции `FORMAT_STRING` возвращается следующий результат:

```
Formatted String
-----
ORACLE, PL/SQL プログラミング, 基礎編, 第3版
```

В позициях разделения лексем пробелами независимо от языка включается запятая.

Следующий поиск `CONTAINS` использует функцию `FORMAT_STRING`:

```
SELECT score (1) "Rank", title
FROM publication
WHERE contains (short_description, format_string('プログラム'), 1) > 0;
```

Результат выглядит так:

```

Rank      TITLE
-----
12   Oracle SQL*Plus デスクトップリファレンス
```

Использование PL/SQL и Oracle Text позволяет выполнять индексирование и полно-текстовый поиск в данных независимо от набора символов или языка.

Дата и время

До настоящего момента наше обсуждение глобализации было сосредоточено исключительно на строках. Тем не менее проблемы с датой и временем порой создают в ходе локализации ничуть не меньше проблем. Пользователи, работающие с вашей базой данных и веб-сервером, могут находиться на другом конце света, но им все равно нужна точная информация, относящаяся к их часовому поясу, а дата и время должны иметь знакомую структуру.

Основные проблемы, усложняющие работу с датой и временем:

- В мире существует много разных часовых поясов.
- В одних регионах действует летнее время, в других его нет.
- В некоторых локальных контекстах используются разные календари.
- В мире не существует общепринятых правил форматирования даты/времени.

Типы данных временных меток

До выхода Oracle9i работа с датой и временем была достаточно тривиальной. В распоряжении разработчика был тип `DATE` и функция `TO_DATE`. Из-за ограниченных возможностей типа `DATE` разработка глобальных приложений становилась делом утомительным и рутинным. Введение любых поправок на часовой пояс требовало ручных вычислений. К сожалению, если ваше приложение должно работать в Oracle8i и более ранних версиях, этот вариант остается единственным.

А в распоряжении пользователей Oracle9i и последующих версий оказываются все выдающиеся возможности типов данных `TIMESTAMP` и `INTERVAL`, подробно описанных в главе 10. Если вы еще не читали эту главу, я приведу краткую сводку, но рекомендую вернуться и прочитать ее для более подробного понимания темы.

Следующий пример демонстрирует типы данных `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE` и `TIMESTAMP WITH LOCAL TIME ZONE` в действии:

```
DECLARE
  v_date_timestamp      TIMESTAMP ( 3 )           := SYSDATE;
  v_date_timestamp_tz   TIMESTAMP ( 3 ) WITH TIME ZONE := SYSDATE;
  v_date_timestamp_ltz  TIMESTAMP ( 3 ) WITH LOCAL TIME ZONE := SYSDATE;
BEGIN
  DBMS_OUTPUT.put_line ('TIMESTAMP: ' || v_date_timestamp);
  DBMS_OUTPUT.put_line ('TIMESTAMP WITH TIME ZONE: ' || v_date_timestamp_tz);
  DBMS_OUTPUT.put_line ( 'TIMESTAMP WITH LOCAL TIME ZONE: '
                        || v_date_timestamp_ltz
                        );
END;
```

Пример возвращает следующие результаты:

```
TIMESTAMP:                08-JAN-05 07.28.39.000 PM
TIMESTAMP WITH TIME ZONE:  08-JAN-05 07.28.39.000 PM -07:00
TIMESTAMP WITH LOCAL TIME ZONE: 08-JAN-05 07.28.39.000 PM
```

Данные `TIMEZONE` и `TIMEZONE WITH LOCAL TIMESTAMP` идентичны, потому что время базы данных принадлежит тому же локальному контексту, что и время сеанса. Значение `TIMESTAMP WITH TIMEZONE` показывает, что пользователь находится в часовом поясе Монтаны. Если бы подключение к базе данных в Колорадо осуществлялось из Калифорнии, то результат был бы немного другим:

```
TIMESTAMP:                08-JAN-05 07.28.39.000 PM
TIMESTAMP WITH TIME ZONE:  08-JAN-05 07.28.39.000 PM -07:00
TIMESTAMP WITH LOCAL TIME ZONE: 08-JAN-05 06.28.39.000 PM
```

Значение `TIMESTAMP WITH LOCAL TIMEZONE` использовало часовой пояс сеанса (смещение `-08:00`), а значение было автоматически преобразовано.

Форматирование даты и времени

Одна из проблем локализации связана с форматированием даты и времени. Например, в Японии принят формат *yyyy/MM/dd hh:mi:ssxff AM*, тогда как в США распространен формат *dd-MON-yyyy hh:mi:ssxff AM*.

Подобные проблемы обычно решаются включением списка форматных масок в таблицу локальных контекстов. При подключении пользователя по его локальному контексту устанавливается правильный формат даты/времени для его региона.

В схему `g11n` включены таблицы `USERS` и `LOCALS`, объединяемые по значению `locale_id`. Рассмотрим несколько примеров использования функций даты/времени (см. главу 10) и форматных масок из таблицы `g11n.locale`.

Столбец `registration_date` в таблице относится к типу данных `TIMESTAMP WITH TIME ZONE`. Функция `TO_CHAR` и передача маски для локального контекста каждого пользователя обеспечивают отображение данных в правильном формате.

```
/* Файл в Сети: g11n.sql */
FUNCTION date_format_func
RETURN sys_refcursor
IS
    v_date    sys_refcursor;
BEGIN
    OPEN v_date
    FOR
        SELECT locale.locale_desc "Locale Description",
               TO_CHAR (users.registration_date,
                       locale.DATE_FORMAT
                       ) "Registration Date"
        FROM users, locale
        WHERE users.locale_id = locale.locale_id;
    RETURN v_date;
END date_format_func;
```

Выполнение функции:

```
variable v_format refcursor
CALL date_format_func() INTO :v_format;
PRINT v_format
```

Результат выглядит так:

| Locale Description | Registration Date |
|--------------------|--|
| English | 01-JAN-2005 11:34:21.000000 AM US/MOUNTAIN |
| Japanese | 2005/01/01 11:34:21.000000 AM JAPAN |
| German | 01 January 05 11:34:21.000000 AM EUROPE/WARSAW |

В каждом из трех локальных контекстов назначена своя маска формата даты. Этот метод позволяет каждому пользователю работать с датой в формате, соответствующем локальному контексту из его профиля. Если теперь добавить условие `NLS_DATE_LANGUAGE`, то при выводе даты и времени будет использоваться соответствующий язык из локального контекста. Давайте проверим правильность вывода на наших таблицах:

```
/* Файл в Сети: g11n.sql */
FUNCTION date_format_lang_func
RETURN sys_refcursor
IS
    v_date    sys_refcursor;
```



```

BEGIN
  OPEN v_date
  FOR
    SELECT locale.locale_desc "Locale Description",
           TO_CHAR (users.registration_date,
                    locale.DATE_FORMAT,
                    'NLS_DATE_LANGUAGE= ' || locale_desc
                    ) "Registration Date"
    FROM users, locale
    WHERE users.locale_id = locale.locale_id;
  RETURN v_date;
END date_format_lang_func;

```

При выполнении функции:

```

variable v_format refcursor
CALL date_format_lang_func() INTO :v_format;
PRINT v_format

```

мы получаем следующий результат:

| Locale Description | Registration Date |
|--------------------|---|
| English | 01-JAN-2005 11:34:21.000000 AM US/MOUNTAIN |
| Japanese | 2005/01/01 11:34:21.000000 午前 JAPAN |
| German | 01 Januar 05 11:34:21.000000 AM EUROPE/WARSAW |

Данные в таблице USERS не изменились, но время выводится в формате конкретного локального контекста. Значение NLS_DATE_LANGUAGE адаптируется для каждой территории, так что для японского локального контекста обозначение полудня (AM) выводится на японском, а название месяца в немецком локальном контексте выводится на немецком.

Функцию можно усовершенствовать так, чтобы в выходных данных учитывался часовой пояс сеанса, — для этого значение преобразуется либо к типу данных `TIMESTAMP WITH TIME ZONE`, либо к локальному часовому поясу сеанса `TIMESTAMP WITH LOCAL TIME ZONE`. Для этого мы воспользуемся функцией `CAST` (см. главу 7), изменяющей тип данных значения, хранящегося в таблице:

```

/* Файл в Сети: g11n.sql */
FUNCTION date_ltz_lang_func
  RETURN sys_refcursor
IS
  v_date sys_refcursor;
BEGIN
  OPEN v_date
  FOR
    SELECT locale.locale_desc,
           TO_CHAR
             (CAST
              (users.registration_date AS TIMESTAMP WITH LOCAL TIME ZONE
              ),
              locale.DATE_FORMAT,
              'NLS_DATE_LANGUAGE= ' || locale_desc
              ) "Registration Date"
    FROM users, locale
    WHERE users.locale_id = locale.locale_id;

  RETURN v_date;
END date_ltz_lang_func;

```

Выполнение функции:

```

variable v_format refcursor
CALL date_ltz_lang_func() INTO :v_format;
PRINT v_format

```

Данные выводятся в следующем формате:

| Locale Description | Registration Date |
|--------------------|--|
| ----- | ----- |
| English | 01-JAN-2005 11:34:21.000000 AM -07:00 |
| Japanese | 2004/12/31 07:34:21.000000 午後 -07:00 |
| German | 01 Januar 05 03:34:21.000000 AM -07:00 |

Обратите внимание на следующие моменты:

- Язык даты/времени преобразуется к локальному контексту.
- Форматирование привязано к локальному контексту.
- Я использую CAST для преобразования значений, хранящихся в формате `TIMESTAMP WITH TIMEZONE`, в `TIMESTAMP WITH LOCAL TIMEZONE`.
- Время выводится относительно часового пояса сеанса (-07:00 в нашем примере).

Во многих примерах, приводившихся ранее, часовой пояс выводился в виде смещения UTC. Такой формат не всегда оказывается самым понятным для пользователя. Oracle ведет список названий регионов и сокращений, которые можно подставлять с изменением маски форматирования. Я вставил три записи, используя имена регионов вместо смещения UTC. Чтобы получить полный список часовых поясов, обратитесь с запросом к представлению `V$TIMEZONE_NAMES`. Другие примеры использования имен регионов встречаются в командах `INSERT` для таблицы `USERS` схемы `g11n`.

Необходимо упомянуть еще один параметр NLS, относящийся к дате/времени. При сохранении времени в таблице из нашего примера используется григорианский календарь, определяемый значением параметра `NLS_CALENDAR`. Однако не все локальные контексты используют тот же календарь, и смена базового календаря не компенсируется никаким форматированием. Параметр `NLS_CALENDAR` позволяет переключиться с григорианского календаря на другие стандартные календари — скажем, на японский имперский календарь:

```
ALTER SESSION SET NLS_CALENDAR = 'JAPANESE IMPERIAL';
ALTER SESSION SET NLS_DATE_FORMAT = 'E RR-MM-DD';
```

После изменения конфигурации сеанса выполняется следующая команда `SELECT`:

```
SELECT sysdate
FROM dual;
```

Она выводит измененное значение `SYSDATE`:

```
SYSDATE
-----
N 17-02-08
```



Значения по умолчанию определяются настройками NLS. Если в системе имеется основной локальный контекст, с которым вы работаете, то задать параметры NLS для базы данных намного проще, чем явно указывать их в приложении. С другой стороны, в приложениях, в которых настройки NLS должны изменяться динамически, рекомендуется включить их в состав настроек пользователя/локального контекста и сохранить их в приложении.

Преобразования денежных величин

Обсуждение глобализации и локализации не было бы полным без упоминания проблем, связанных с конвертацией валют. Самый распространенный способ перевода, скажем, долларов в иены, основан на использовании таблицы курсов. Но откуда приложение узнает, как должны отображаться полученные числа?

- Могут ли в них присутствовать точки и запятые и где они должны располагаться?
- Каким символом должна обозначаться каждая валюта (\$ для доллара, € для евро и т. д.)?
- Какое обозначение ISO следует использовать (USD, например)?

Каждое печатное издание в схеме `g11n` имеет цену и связывается с локальным контекстом. Для форматирования строк можно воспользоваться функцией `TO_CHAR`, но как насчет правильного отображения цены? Можно воспользоваться параметром `NLS_CURRENCY`:

```
/* Файл в Сети: g11n.sql */
FUNCTION currency_conv_func
RETURN sys_refcursor
IS
    v_currency sys_refcursor;
BEGIN
    OPEN v_currency
    FOR
        SELECT pub.title "Title",
               TO_CHAR (pub.price,
                       locale.currency_format,
                       'NLS_CURRENCY=' || locale.currency_symbol
                       ) "Price"
        FROM publication pub, locale
        WHERE pub.locale_id = locale.locale_id;

    RETURN v_currency;
END currency_conv_func;
```

При выполнении функции преобразования:

```
VARIABLE v_currency REFCURSOR
CALL currency_conv_func() INTO :v_currency;
PRINT v_currency
```

выводится следующий список цен:

| Title | Price |
|--|---------|
| ----- | ----- |
| Oracle PL/SQL Programming, 3rd Edition | \$54.95 |
| Oracle PL/SQL プログラミング 基礎編 第3版 | ¥5,800 |
| Oracle PL/SQL Programmierung, 2. Auflage | €64 |

Обратите внимание: собственно преобразование в этом примере не выполняется. Если вам понадобится автоматизировать преобразование денежных сумм из одной валюты в другую, определите таблицу курсов и правила преобразования.

Обозначение `NLS_ISO_CURRENCY` обычно представляет собой сокращенное название валюты из трех символов. Если не считать немногочисленных исключений, первые два символа определяют страну или локальный контекст, а третий символ — собственно валюту. Например, американский доллар и японская иена обозначаются соответственно `USD` и `JPY`. Впрочем, евро сейчас используется во многих европейских странах, так что это правило уже не работает — обозначение `EUR` используется независимо от страны.

В схему `g11n` включены данные, которые упрощают вывод цен печатных изданий с правильными сокращениями ISO. Пример такого рода встречается в функции `ISO_CURRENCY_FUNC`:

```
/* Файл в Сети: g11n.sql */
FUNCTION iso_currency_func
RETURN sys_refcursor
IS
    v_currency sys_refcursor;
BEGIN
    OPEN v_currency
```

```

FOR
    SELECT    title "Title",
              TO_CHAR (pub.price,
                      locale.iso_currency_format,
                      'NLS_ISO_CURRENCY=' || locale.iso_currency_name
                      ) "Price - ISO Format"
    FROM publication pub, locale
    WHERE pub.locale_id = locale.locale_id
    ORDER BY publication_id;

RETURN v_currency;
END iso_currency_func;

```

Выполнение функции ISO_CURRENCY_FUNC:

```

VARIABLE v_currency REFCURSOR
CALL iso_currency_func() INTO :v_currency;
PRINT v_currency

```

Результат:

| Title | Price - ISO Format |
|--|--------------------|
| Oracle PL/SQL Programming, 3rd Edition | USD54.95 |
| Oracle PL/SQLプログラミング 基礎編 第3版 | JPY5,800 |
| Oracle PL/SQL Programmierung, 2. Auflage | EUR64 |

Обозначения USD, JPY и EUR включаются в выводимые цены, как и положено в соответствии с маской формата.

Globalization Development Kit для PL/SQL

В Oracle10g впервые появился инструментарий Globalization Development Kit (GDK) для Java и PL/SQL, упрощающий процесс разработки приложений с поддержкой глобализации. Если вы занимаетесь разработкой многоязычного приложения, определение локального контекста каждого пользователя и вывод информации в соответствии с требованиями локального контекста может стать самой сложной задачей программирования, которую вам предстоит решить. Входящие в GDK компоненты PL/SQL разделены на два пакета: UTL_I18N и UTL_LMS.

Пакет UTL_I18N

Пакет UTL_I18N содержит основную рабочую функциональность GDK. Его подпрограммы перечислены в табл. 25.7.

Функция GET_LOCAL_LANGUAGES является одной из самых полезных в этом пакете. Зная территорию пользователя, можно дополнительно сократить список значений при помощи функции UTL_I18N.GET_LOCAL_LANGUAGES. Данная возможность замечательно подходит для приложений, в которых администратор должен настраивать конфигурацию для конкретных пользователей. Для демонстрации этой функции будут использоваться следующие данные:

```

CREATE TABLE user_admin (
    id NUMBER(10) PRIMARY KEY,
    first_name VARCHAR2(10 CHAR),
    last_name VARCHAR2(20 CHAR),
    territory VARCHAR2(30 CHAR),
    language VARCHAR2(30 CHAR))
/

```

```

BEGIN
INSERT INTO user_admin
VALUES (1, 'Stan', 'Smith', 'AMERICA', 'AMERICAN');
INSERT INTO user_admin
VALUES (2, 'Robert', 'Hammon', NULL, 'SPANISH');
INSERT INTO user_admin
VALUES (3, 'Anil', 'Venkat', 'INDIA', NULL);
COMMIT;
END;
/

```

Таблица 25.7. Подпрограммы пакета UTL_I18N

| Имя | Описание |
|-----------------------------|---|
| ESCAPE_REFERENCE | Документы HTML и XML не всегда поддерживают те же символы, что и база данных. В таких случаях бывает полезно заменить неподдерживаемые символы служебными комбинациями (escapes). Функция получает входную строку и набор символов документа HTML или XML |
| GET_COMMON_TIME_ZONES | Возвращает список основных часовых поясов. Функция особенно удобна для вывода списка, из которого пользователь выбирает часовой пояс для своего профиля |
| GET_DEFAULT_CHARSET | Возвращает имя набора символов по умолчанию |
| GET_DEFAULT_ISO_CURRENCY | Получает обозначение региона и возвращает соответствующий код валюты |
| GET_DEFAULT_LINGUISTIC_SORT | Возвращает самый распространенный критерий сортировки для заданного языка |
| GET_LOCAL_LANGUAGES | Возвращает локальные языки для заданной территории |
| GET_LOCAL_LINGUISTIC_SORTS | Возвращает список имен критериев сортировки для заданного языка |
| GET_LOCAL_TERRITORIES | Возвращает список территорий для заданного языка |
| GET_LOCAL_TIMEZONES | Возвращает все часовые пояса для заданной территории |
| GET_TRANSLATION | Переводит название языка и/или территории на заданный язык и возвращает результат |
| MAP_CHARSET | Особенно полезна в приложениях, которые пересылают данные, извлеченные из базы, по электронной почте. Обеспечивает преобразование между набором символов базы данных и набором символов, безопасным для пересылки по электронной почте |
| MAP_FROM_SHORT_LANGUAGE | Для полученного сокращенного названия языка возвращает полное название языка в Oracle |
| MAP_LANGUAGE_FROM_ISO | Получает имя локального контекста в стандарте ISO и возвращает название языка Oracle |
| MAP_LOCALE_TO_ISO | Получает язык и территорию, возвращает имя локального контекста в стандарте ISO |
| MAP_TERRITORY_FROM_ISO | Получает локальный контекст ISO, возвращает название территории Oracle |
| MAP_TO_SHORT_LANGUAGE | Функция, обратная по отношению к MAP_FROM_SHORT_LANGUAGE: получает полное название языка Oracle, возвращает короткое название |
| RAW_TO_CHAR | Перегруженная функция, которая получает тип RAW и возвращает VARCHAR2 |
| RAW_TO_NCHAR | Функция идентична RAW_TO_CHAR, но возвращает значение типа NVARCHAR2 |
| STRING_TO_RAW | Преобразует VARCHAR2 или NVARCHAR2 к заданному набору символов, возвращает значение типа RAW |
| TRANSLITERATE | Возвращает транслитерацию строки, записанной японской каной |
| UNESCAPE_REFERENCE | Функция, обратная по отношению к ESCAPE_REFERENCE: распознает служебные комбинации и преобразует их в исходные символы |

Территория вводится в таблице USER_ADMIN. Вывод списка языков для пользователя Anil осуществляется следующим анонимным блоком:

```

DECLARE
-- Создание массива для результирующего набора языков
v_array utl_i18n.string_array;
-- Создание переменной для хранения данных пользователя
v_user user_admin%ROWTYPE;
BEGIN
-- Заполнение переменной данными пользователя Anil
SELECT *
  INTO v_user
  FROM user_admin
 WHERE ID = 3;

-- Получение списка языков для территории
v_array := utl_i18n.get_local_languages (v_user.territory);
DBMS_OUTPUT.put (CHR (10));
DBMS_OUTPUT.put_line ('=====');
DBMS_OUTPUT.put_line ('User: ' || v_user.first_name || ' '
                      || v_user.last_name
                      );
DBMS_OUTPUT.put_line ('Territory: ' || v_user.territory);
DBMS_OUTPUT.put_line ('=====');

-- Перебор элементов массива
FOR y IN v_array.FIRST .. v_array.LAST
LOOP
  DBMS_OUTPUT.put_line (v_array (y));
END LOOP;
END;
```

Программа возвращает следующий результат:

```

=====
User: Anil Venkat
Territory: INDIA
=====
ASSAMESE
BANGLA
GUJARATI
HINDI
KANNADA
MALAYALAM
MARATHI
ORIYA
PUNJABI
TAMIL
TELUGU
```

Работать с таким списком намного удобнее, чем с полным списком всех языков. Аналогичная фильтрация может выполняться и для территорий в том случае, если известен язык. Допустим, у пользователя **Robert** в данный момент территория не определена, но задан испанский язык (**SPANISH**). Следующий анонимный блок возвращает список действительных территорий для испанского языка:

```

DECLARE
-- Создание массива для результирующего набора территорий
v_array utl_i18n.string_array;
-- Создание переменной для хранения данных пользователя
v_user user_admin%ROWTYPE;
BEGIN
-- Заполнение переменной данными пользователя Robert
SELECT *
  INTO v_user
  FROM user_admin
 WHERE ID = 2;
```

```

-- Получение списка территорий для языка
v_array := utl_i18n.get_local_territories (v_user.LANGUAGE);
DBMS_OUTPUT.put (CHR (10));
DBMS_OUTPUT.put_line ('=====');
DBMS_OUTPUT.put_line ('User: ' || v_user.first_name || ' '
                      || v_user.last_name
                      );
DBMS_OUTPUT.put_line ('Language: ' || v_user.LANGUAGE);
DBMS_OUTPUT.put_line ('=====');

-- Перебор элементов массива
FOR y IN v_array.FIRST .. v_array.LAST
LOOP
    DBMS_OUTPUT.put_line (v_array (y));
END LOOP;
END;
```

Результат:

```

=====
User: Robert Hammon
Language: SPANISH
=====
SPAIN
CHILE
COLOMBIA
COSTA RICA
EL SALVADOR
GUATEMALA
MEXICO
NICARAGUA
PANAMA
PERU
PUERTO RICO
VENEZUELA
```

Зная территорию, можно вывести список языков, действительных часовых поясов и валют; все это значительно упростит настройку конфигурации. Если выбран язык, средства UTL_I18N позволяют получить для него набор символов по умолчанию, лингвистическую сортировку по умолчанию, локальные территории и короткое название языка.

Пакет обработки ошибок UTL_LMS

UTL_LMS — второй пакет, входящий в GDK. В него включены две функции для выборки и форматирования сообщений об ошибках:

- GET_MESSAGE — возвращает необработанное сообщение об ошибке для заданного языка (иначе говоря, параметры сообщения не включены в возвращаемый текст).
- FORMAT_MESSAGE — включает в сообщение дополнительную информацию.

Пример:

```

DECLARE
    v_bad_bad_variable    PLS_INTEGER;
    v_function_out        PLS_INTEGER;
    v_message             VARCHAR2 (500);
BEGIN
    v_bad_bad_variable := 'x';
EXCEPTION
    WHEN OTHERS
    THEN
        v_function_out :=
            utl_lms.GET_MESSAGE (06502, 'rdbms', 'ora', NULL, v_message);
-- Вывод неформатированных и отформатированных сообщений
```

продолжение ➤

```

DBMS_OUTPUT.put (CHR (10));
DBMS_OUTPUT.put_line ('Message - Not Formatted');
DBMS_OUTPUT.put_line ('=====');
DBMS_OUTPUT.put_line (v_message);
DBMS_OUTPUT.put (CHR (10));
DBMS_OUTPUT.put_line ('Message - Formatted');
DBMS_OUTPUT.put_line ('=====');
DBMS_OUTPUT.put_line (utl_lms.format_message (v_message,
                                              ': The quick brown fox'
                                              )
                      );

```

END;

При вызове UTL_LMS.GET_MESSAGE язык не был задан, поэтому сообщение возвращается на языке по умолчанию, определяемом параметром NLS_LANGUAGE.

```

Message - Not Formatted
=====
PL/SQL: numeric or value error%

```

```

Message - Formatted
=====
PL/SQL: numeric or value error: The quick brown fox

```

Так как при вызове UTL_LMS.GET_MESSAGE может передаваться язык, при получении сообщения я просто передаю язык пользователя приложения.

Варианты реализации GDK

Функции GDK позволяют выбрать несколько разных вариантов реализации. Если в системе должны поддерживаться только два-три локальных контекста, возможно, реализацию проще всего разделить по контекстам. Скажем, для немецкой системы серверы базы данных и приложений настраиваются для этого локального контекста, а для пользователей из Франции используется совершенно иная среда. Однако чаще требуется реализовать полноценную многоязычную среду, в которой новые контексты могут добавляться без приобретения и настройки отдельного экземпляра системы. Такая реализация потребует дополнительных усилий на начальной стадии, но ею гораздо проще управлять в долгосрочной перспективе.

Метод определения локального контекста пользователя в значительной степени зависит от предполагаемой аудитории и типа разрабатываемого приложения. В следующих подразделах рассматриваются три возможных решения, которые следует принять во внимание при проектировании.

Метод 1. Кнопки локальных контекстов

Этот метод часто встречается на веб-страницах в Интернете. Посетите сайт компании, которая ведет бизнес в разных странах; на главной странице часто размещаются кнопки или ссылки для выбора языка:

in English | en Español | en Français | in Italiano

Данное решение идеально подходит для ограниченного набора локальных контекстов. Пользователь выбирает свой язык и денежную единицу простым щелчком на ссылке, а если при выборе будет допущена ошибка — достаточно щелкнуть на кнопке **Назад** в браузере и исправить проблему.

В этом сценарии либо для каждого локального контекста создаются отдельные страницы и код, либо настройки для текущего сеанса сохраняются в cookie или базе данных,

чтобы локализация проходила под контролем базы данных. Чаще всего управление локализацией выполняется на уровне приложения.

Метод 2. Администрирование пользователей

Метод 2 отлично подходит для управления настройками локального контекста в приложениях с управляемой пользовательской базой (то есть закрытых, например, для анонимных пользователей Интернета). При небольшом количестве пользователей можно воспользоваться пакетом `UTL_I18N` для вывода списка доступных часовых поясов, локальных контекстов, языков и территорий, как было показано ранее. Пользователь или администратор просто выбирает настройки, подходящие для пользователя; при каждом входе пользователя приложение читает эти настройки и выполняет соответствующую локализацию.

А если количество пользователей очень велико? Управлять настройками каждого пользователя по отдельности в этом случае нереально. Можно, по примеру проектировщиков базы данных Oracle, организовать систему профилей. Реализуйте в своем приложении возможность создания профиля и настройки локального контекста на уровне профиля, а не на уровне пользователя. При добавлении нового пользователя просто назначьте ему профиль. Тем самым вы избавитесь от многих административных хлопот, особенно если позднее вдруг понадобится внести изменения в конфигурацию. Вместо того чтобы изменять данные всех пользователей, достаточно изменить профиль.

Метод 3. Гибридный

Метод 3 сочетает в себе отдельные аспекты методов 1 и 2. Он часто применяется в приложениях интернет-магазинов. Большинство покупателей начинают с просмотра сайта в поисках нужного товара. На этой стадии требовать от них ввода полной информации об их местонахождении преждевременно, но при этом данные должны сортироваться в правильном порядке и выводиться на нужном языке, с правильным форматом денежных величин. Чтобы обеспечить правильность базовой информации локального контекста, предложите решение, описанное в методе 1.

А когда покупатель примет решение о покупке, вы требуете, чтобы он ввел профильные данные с информацией о локальном контексте. Локализация становится более конкретной, в ней используются точная дата/время и финансовая информация из базы данных.

26

Объектно-ориентированные возможности PL/SQL

Язык PL/SQL всегда поддерживал традиционные процедурные стили программирования, в частности структурное проектирование и функциональную декомпозицию. Пакеты PL/SQL позволяют использовать также объектно-ориентированный подход, применяя в работе с реляционными таблицами принципы абстракции и инкапсуляции. В новых версиях Oracle введена непосредственная поддержка объектно-ориентированного программирования (ООП). Программистам стали доступны богатая и сложная система типов, иерархия, а также взаимозаменяемость типов.

Хотя тема объектно-ориентированного программирования в Oracle могла бы стать предметом отдельной книги, мы рассмотрим лишь несколько примеров, демонстрирующих важнейшие аспекты объектно-ориентированного программирования на PL/SQL:

- создание и использование объектных типов;
- наследование и взаимозаменяемость;
- эволюция типов;
- выборка данных на основе REF-ссылок;
- объектные представления, в том числе `INSTEAD OF`.

Не рассчитывайте найти в этой главе:

- Полные диаграммы синтаксиса команд SQL для работы с объектными типами.
- Обсуждение вопросов, относящихся к компетенции администраторов базы данных, — например, импортирования и экспортирования объектных данных.
- Описания низкоуровневых аспектов (структуры хранения данных на диске).

Начнем с краткого исторического экскурса.

История объектных возможностей Oracle

Впервые появившиеся в 1997 году как дополнение к Oracle8, объектные возможности позволили разработчикам расширить набор встроенных типов данных Oracle *абстрактными типами* данных. Также в Oracle8 были введены определяемые программистом коллекции (см. главу 12), оказавшиеся очень удобными. Объектная модель Oracle обеспечивает много интересных возможностей, в частности доступ к данным через указатели, но она не поддерживает ни наследования, ни динамического полиморфизма, и поэтому объектно-ориентированные средства Oracle8 вряд ли произведут впечатление

на приверженцев настоящего ООП. Сложность и низкая производительность объектных функций также не способствуют их успеху.

В Oracle8i была введена поддержка хранимых процедур Java, которые позволяли программировать на менее специализированном языке, чем PL/SQL, и упростили разработку хранимых процедур для сторонников ООП. Появился способ преобразования объектных типов, определенных на сервере, в Java-классы, что делало возможным совместную работу с данными в Java и в базе данных. Версия Oracle8i вышла в период наивысшего интереса к языку Java, поэтому мало кто заметил, что объектные функции Oracle почти не изменились, разве что начали понемногу интегрироваться с базовым сервером. В то время я спросил одного из представителей Oracle о будущем ООП на языке PL/SQL, и тот ответил: «Если вам требуется настоящее объектно-ориентированное программирование, пользуйтесь Java».

Однако в Oracle9i встроенная поддержка объектов была значительно расширена. Введена поддержка наследования и полиморфизма в базах данных, PL/SQL был оснащен новыми объектными средствами. Имеет ли смысл расширять объектную модель системы на структуру базы данных? Следует ли переписать существующие приложения клиентского и промежуточного уровней? Как показано в табл. 26.1, в Oracle были реализованы значительные достижения в ООП, и переход на эту технологию выглядит очень заманчиво. Также в таблице перечислены полезные возможности, которые еще не реализованы¹.

Таблица 26.1. Возможности Oracle

| Средства и возможности | 8.0 | 8.1 | 9.1 | 9.2 и выше | 11g и выше |
|--|-----|-----|-----|------------|------------|
| Абстрактные типы данных как равноправные сущности базы данных | * | * | * | * | * |
| Абстрактные типы данных как параметры PL/SQL | * | * | * | * | * |
| Атрибуты с типами коллекций | * | * | * | * | * |
| Атрибуты типа REF для навигации по объектам внутри базы данных | * | * | * | * | * |
| Реализация логики методов на PL/SQL и C | * | * | * | * | * |
| Определяемая программистом семантика сравнения объектов | * | * | * | * | * |
| Представление реляционных данных как данных объектных типов | * | * | * | * | * |
| Статический полиморфизм (перегрузка методов) | * | * | * | * | * |
| Возможность «эволюции» типа путем модификации логики существующих методов или добавления новых методов | * | * | * | * | * |
| Реализация логики методов на языке Java | | * | * | * | * |
| «Статические» методы (уровня класса, а не уровня экземпляра) | | * | * | * | * |
| Возможность использования первичного ключа как идентификатора хранимого объекта, обеспечивающая декларативную целостность REF-ссылок | | * | * | * | * |
| Наследование атрибутов и методов от пользовательских типов | | | * | * | * |
| Динамическая диспетчеризация методов | | | * | * | * |
| Супертипы, для которых не могут создаваться экземпляры (аналоги абстрактных классов языка Java) | | | * | * | * |
| Возможность эволюции типа путем удаления методов (и добавления для изменения сигнатуры) | | | * | * | * |

¹ Возможно, правильнее будет сказать «предположительно полезные» возможности. Вряд ли они произведут революцию.

Таблица 26.1 (продолжение)

| Средства и возможности | 8.0 | 8.1 | 9.1 | 9.2 и выше | 11g и выше |
|---|-----|-----|-----|------------|------------|
| Возможность эволюции типа путем добавления и удаления атрибутов, автоматическое распространение изменений на связанные структуры физической базы данных | | | * | * | * |
| «Анонимные» типы: ANYTYPE, ANYDATA, ANYDATASET | | | * | * | * |
| Операторы понижающего преобразования (TREAT) и определения типа (IS OF), доступные в SQL | | | * | * | * |
| Операторы TREAT и IS OF в PL/SQL | | | | * | * |
| Определяемые пользователем конструкторы | | | | * | * |
| Вызовы методов супертипа в производном типе | | | | | * |
| Приватные атрибуты, переменные, константы и методы | | | | | |
| Наследование от нескольких супертипов | | | | | |
| Совместное использование объектных типов или экземпляров в распределенных базах данных без обращения к объектным представлениям | | | | | |



В Oracle Database 10g было включено несколько полезных улучшений в области коллекций (см. главу 12), но только одна новая возможность, относящаяся к объектным типам: она описана во врезке «Псевдостолбец OBJECT_VALUE» (см. с. 936).

Если вы еще не применяете объектно-ориентированное программирование в своих разработках, многие термины в этой таблице покажутся вам незнакомыми. Однако из оставшейся части этой главы вы поймете их смысл и получите представление о более масштабных архитектурных решениях, которые вам придется принимать.

Пример объектного приложения

Идея для примера этой главы взята из книги *Learning Oracle PL/SQL* (изд-во O'Reilly). Мы построим систему на базе Oracle, в которой объектно-ориентированный подход используется для реализации библиотечного каталога. В каталоге хранится информация о книгах, периодических изданиях (журналах и газетах) и других печатных изданиях.



Рис. 26.1. Иерархия типов библиотечного каталога

Графическое представление типов верхнего уровня каталога можно увидеть на рис. 26.1. В дальнейшем иерархия будет дополнена объектами, обозначенными пунктиром.

Создание базового типа

Корневой элемент (вершина иерархии) представляет общие характеристики всех подтипов. Допустим, что у книг и периодики имеются только две общие характеристики: библиотечный идентификатор и название. Таким образом, объектный тип для элемента каталога создается в SQL*Plus следующей командой SQL:

```
CREATE OR REPLACE TYPE catalog_item_t AS OBJECT (  
    id INTEGER,  
    title VARCHAR2(4000),  
    NOT INSTANTIABLE MEMBER FUNCTION ck_digit_okay  
        RETURN BOOLEAN,  
    MEMBER FUNCTION print  
        RETURN VARCHAR2  
) NOT INSTANTIABLE NOT FINAL;
```

Команда создает объектный тип — аналог класса Java или C++. Если провести аналогию с реляционной моделью, объектный тип подобен типу записи с набором связанных с ним функций и процедур (называемых *методами*).

Ключевые слова `NOT FINAL` в конце определения типа указывают, что он может служить *базовым* или *супертипом*, то есть на его основе могут определяться другие типы. В данном случае мы собираемся создать подтипы для книг и периодических изданий; если опустить эти ключевые слова, по умолчанию Oracle использует слово `FINAL`, запрещающее создание подтипов на основе данного типа.

Обратите внимание на то, что спецификация данного типа содержит предложение `NOT INSTANTIABLE`. Это означает, что PL/SQL позволит объявлять переменные типа `catalog_item_t`, но им нельзя будет присваивать значения — ни явно, ни другим способом. Подобно *абстрактному классу* Java, данный тип предназначен исключительно для использования в качестве основы для создания подтипов (для которых, конечно, можно будет создавать и экземпляры объектов).

Для удобства в тип включен метод `print` (кстати, это *не* зарезервированное слово), который позволяет получить описание объекта в виде одной строки. При создании подтипа метод будет перегружен, то есть подтип будет содержать метод с тем же именем, но возвращающий атрибуты подтипа. Обратите внимание: вместо оформления `print` в виде процедуры, из-за чего в программе было бы зафиксировано использование `DBMS_OUTPUT.PUT_LINE` или чего-нибудь в этом роде, я решил использовать функцию, вывод которой можно будет позднее перенаправить. Такое решение не является объектно-ориентированным; просто признак хорошего проектирования.

Также определяется метод `ck_digit_okay`, который возвращает `TRUE` или `FALSE` в зависимости от того, совпала ли контрольная цифра. Предполагается, что все объекты подтипов `catalog_item_t` будут содержать идентификаторы, включающие контрольную цифру для проверки правильности записи. Поскольку мы собираемся работать только с книгами и периодикой, которые обычно идентифицируются кодами ISBN и ISSN, то к этим подтипам применима концепция контрольной цифры.

Прежде чем переходить к следующей части примера, обратите внимание на некоторые обстоятельства:

- Приведенная выше команда `CREATE TYPE` создает спецификацию объектного типа. Соответствующее тело типа с реализацией методов создается отдельно командой `CREATE TYPE BODY`.

- Для объектных типов используется то же пространство имен, что для таблиц и программ PL/SQL верхнего уровня. Это одна из причин, по которым в именах типов используется префикс «_t».
- Объектные типы всегда принадлежат создавшему их пользователю (схеме) Oracle, который может предоставлять привилегию EXECUTE для них другим пользователям.
- Синонимы объектных типов поддерживаются только начиная с Oracle9i Release 2 и выше.
- Как и традиционные программы PL/SQL, объектные типы можно определять с разрешениями создателя (по умолчанию) или вызывающего (см. главу 24).
- В отличие от объектных моделей некоторых языков, в объектной модели Oracle отсутствует корневой класс, от которого образуются все остальные пользовательские классы.
- Если вы столкнетесь с ошибкой компиляции PLS-00103, вероятно, вы допустили распространенную ошибку и завершили метод символом «;». В этой спецификации типа должна использоваться запятая.

Создание подтипа

Поскольку для типа `catalog_item_t` не могут создаваться экземпляры, нужно определить его подтипы. Начнем с подтипа для объектов-книг. Так как книга является одним из элементов каталога, в нашем примере все экземпляры подтипа `book_t` будут обладать четырьмя атрибутами:

- `id` — наследуется от базового типа `catalog_item_t`;
- `title` — также наследуется от базового типа;
- `isbn` — соответствует коду ISBN книги;
- `pages` — количество страниц книги.

Соответствующий программный код выглядит так:

```

1  TYPE book_t UNDER catalog_item_t (
2      isbn VARCHAR2(13),
3      pages INTEGER,
4
5      CONSTRUCTOR FUNCTION book_t (id IN INTEGER DEFAULT NULL,
6          title IN VARCHAR2 DEFAULT NULL,
7          isbn IN VARCHAR2 DEFAULT NULL,
8          pages IN INTEGER DEFAULT NULL)
9          RETURN SELF AS RESULT,
10
11      OVERRIDING MEMBER FUNCTION ck_digit_okay
12          RETURN BOOLEAN,
13
14      OVERRIDING MEMBER FUNCTION print
15          RETURN VARCHAR2
16  );
```

Интересные части кода описаны в следующей таблице.

| Строки | Описание |
|--------|--|
| 1 | Для определения подтипа используется ключевое слово UNDER. Конструкция AS OBJECT здесь не используется, потому что она будет лишней: производным от объектного типа может быть только другой объектный тип |
| 2–3 | Перечисляются только те атрибуты, которые уникальны для подтипа; атрибуты родительского типа включаются в подтип автоматически. Oracle сначала создает атрибуты базового типа, а затем атрибуты подтипа в порядке их следования в спецификации |
| 5–15 | Объявления методов (см. следующий раздел) |

Методы

В приведенном выше определении типа используется два вида методов:

- **Конструктор** — функция, которая получает значения всех атрибутов и помещает их в объект заданного типа. Объявлена в строках 5–6.
- **Методы экземпляров** — функции или процедуры, выполняемые в контексте экземпляра объекта, то есть имеющие доступ к текущим значениям атрибутов. Методы экземпляров объявляются в строках 11–12 и 14–15.

В нашем примере продемонстрировано определение пользовательского конструктора (такая возможность появилась в Oracle9i Release 2). В предыдущих версиях Oracle поддерживались только конструкторы, автоматически генерируемые системой. Создание собственного конструктора типа позволяет управлять созданием экземпляров этого типа. В конструкторе можно проверить и инициализировать данные, а также реализовать полезные побочные эффекты. Кроме того, можно определить несколько перегруженных версий пользовательского конструктора для разных вариантов его вызова.

Чтобы увидеть методы и типы в действии, выполните следующий анонимный блок:

```
1  DECLARE
2      generic_item catalog_item_t;
3      abook book_t;
4  BEGIN
5      abook := NEW book_t(title => 'Out of the Silent Planet',
6                          isbn => '0-6848-238-02');
7      generic_item := abook;
8      DBMS_OUTPUT.PUT_LINE('BOOK: ' || abook.print());
9      DBMS_OUTPUT.PUT_LINE('ITEM: ' || generic_item.print());
10 END;
```

Вызовы метода print объектов abook и generic_item (строки 8 и 9) возвращают идентичные результаты:

```
BOOK: id=; title=Out of the Silent Planet; isbn=0-6848-238-02; pages=
ITEM: id=; title=Out of the Silent Planet; isbn=0-6848-238-02; pages=
```

| Строки | Описание |
|--------|--|
| 5–6 | Конструктор строит новый объект и помещает его в переменную. В данном примере аргументы передаются по именам. Мы задали значения только двух атрибутов из четырех, но конструктор все равно создал объект. Синтаксис использования конструктора: [NEW] имя_типа (аргумент1, аргумент2, ...); Ключевое слово NEW, появившееся в Oracle9i Database Release 2, не является обязательным, но оно наглядно указывает на то, что команда создаст новый объект |
| 7 | Хотя экземпляры экземпляра каталога создавать запрещено, представляющей его переменной можно присвоить экземпляр подтипа, и в ее значении будут содержаться атрибуты, уникальные для данного подтипа. Этот факт демонстрирует важный аспект «взаимозаменяемости» типов в PL/SQL: ее суть заключается в том, что в объектной переменной может содержаться экземпляр любого подтипа типа данных этой переменной |
| 8–9 | Для вызова метода print() используется классический точечный синтаксис: объект.имя_метода(аргумент_1, аргумент_2, ...) поскольку этот метод является методом экземпляра и его можно вызвать только для уже объявленного и инициализированного экземпляра объекта. Всегда вызывается метод самого специализированного подтипа (расположенного на самом нижнем уровне иерархии), связанного с текущим экземпляром объекта. Выбор метода откладывается до этапа выполнения — эта технология называется динамической диспетчеризацией методов. Она очень удобна, хотя и ухудшает производительность |

Чтобы лучше понять полученный результат, рассмотрим тело метода `book_t`. В реализации данного типа используется две важные концепции, о которых будет рассказано далее.

```

1  TYPE BODY book_t
2  AS
3      CONSTRUCTOR FUNCTION book_t (id IN INTEGER,
4          title IN VARCHAR2,
5          isbn IN VARCHAR2,
6          pages IN INTEGER)
7          RETURN SELF AS RESULT
8  IS
9      BEGIN
10         SELF.id := id;
11         SELF.title := title;
12         SELF.isbn := isbn;
13         SELF.pages := pages;
14         IF isbn IS NULL OR SELF.ck_digit_okay
15             THEN
16             RETURN;
17         ELSE
18             RAISE_APPLICATION_ERROR(-20000, 'ISBN ' || isbn
19                 || ' has bad check digit');
20         END IF;
21     END;
22
23     OVERRIDING MEMBER FUNCTION ck_digit_okay
24         RETURN BOOLEAN
25     IS
26         subtotal PLS_INTEGER := 0;
27         isbn_digits VARCHAR2(10);
28     BEGIN
29         /* Удаление дефисов и пробелов */
30         isbn_digits := REPLACE(REPLACE(SELF.isbn, '-'), ' ');
31         IF LENGTH(isbn_digits) != 10
32             THEN
33             RETURN FALSE;
34         END IF;
35
36         FOR nth_digit IN 1..9
37             LOOP
38                 subtotal := subtotal +
39                     (11 - nth_digit) *
40                     TO_NUMBER(SUBSTR(isbn_digits, nth_digit, 1));
41             END LOOP;
42
43         /* check digit can be 'X' which has value of 10 */
44         IF UPPER(SUBSTR(isbn_digits, 10, 1)) = 'X'
45             THEN
46             subtotal := subtotal + 10;
47         ELSE
48             subtotal := subtotal + TO_NUMBER(SUBSTR(isbn_digits, 10, 1));
49         END IF;
50
51         RETURN MOD(subtotal, 11) = 0;
52
53     EXCEPTION
54         WHEN OTHERS
55             THEN
56             RETURN FALSE;
57     END;
58
59     OVERRIDING MEMBER FUNCTION print
60         RETURN VARCHAR2
61     IS

```



```

62     BEGIN
63         RETURN 'id=' || id || '; title=' || title
64             || '; isbn=' || isbn || '; pages=' || pages;
65     END;
66 END;
```

Пользовательский конструктор должен подчиняться определенным правилам:

- Он должен быть объявлен с ключевыми словами **CONSTRUCTOR FUNCTION** (строка 3).
- Предложение, определяющее возвращаемое значение, должно выглядеть так: **RETURN SELF AS RESULT** (строка 7).
- Конструктор может присваивать значения любым атрибутам текущего объекта (строки 10–13).
- Он должен завершаться оператором **RETURN** или исключением (строка 16; строки 18–19).

Обычно конструктор присваивает значения большинству атрибутов объекта. Как видно из строки 14, перед окончанием работы конструктор проверяет контрольную цифру. Как видно из строки 30, атрибуты объекта (такие, как **SELF.isbn**) доступны еще до завершения проверки — интересная и полезная возможность.

Код в строках 18–19 представляет собой простую заготовку. В реальном приложении обработка исключений, специфических для конкретного приложения, не столь тривиальна (см. главу 6).

Теперь обратимся к ключевому слову **SELF**, которое несколько раз встречается в коде объектного типа (не только в конструкторе, но и других методах), — аналогу ключевого слова **this** в языке Java. **SELF** — это ссылка на вызывающий (текущий) экземпляр, используемая исключительно в реализации методов. Ее можно применять для ссылки на весь объект, а с точечным синтаксисом — для ссылки на атрибут или метод объекта:

```
IF SELF.id ...
```

```
IF SELF.chk_digit_okay() ...
```

В методах экземпляров ключевое слово **SELF** не обязательно (см. строки 63–64), поскольку идентификаторы текущего объекта всегда видны в методе. В строках 10–13 оно было необходимо — в данном случае имена формальных параметров конструктора совпадают с именами атрибутов, и компилятор идентифицировал бы их как ссылки на параметры. Кроме того, они делают код более понятным.

Еще несколько рекомендаций, касающихся применения ключевого слова **SELF**:

- Ключевое слово **SELF** не может использоваться в статических методах, поскольку у них нет «текущего объекта».
- По умолчанию в функциях ключевое слово **SELF** интерпретируется как входная переменная (**IN**), а в процедурах и конструкторах — как входная и выходная переменная (**IN OUT**).
- Ссылку на текущий объект можно передать явно в первом формальном параметре. Вычисление контрольной цифры в строках 23–57 приведено только для примера. К сожалению, наш алгоритм не использует в полной мере все новые объектно-ориентированные средства. Кроме того, не полностью реализован обработчик исключений. Он должен обрабатывать такие ситуации, как передача функции **TO_NUMBER** строки вместо числа.

Перейдем ко второму подтипу — периодическим изданиям:

```

TYPE serial_t UNDER catalog_item_t (
    issn VARCHAR2(10),
    open_or_closed VARCHAR2(1),
```

продолжение ➤

```

CONSTRUCTOR FUNCTION serial_t (id IN INTEGER DEFAULT NULL,
    title IN VARCHAR2 DEFAULT NULL,
    issn IN VARCHAR2 DEFAULT NULL,
    open_or_closed IN VARCHAR2 DEFAULT NULL)
    RETURN SELF AS RESULT,

OVERRIDING MEMBER FUNCTION ck_digit_okay
    RETURN BOOLEAN,

OVERRIDING MEMBER FUNCTION print
    RETURN VARCHAR2
) NOT FINAL;

```

И снова в спецификации типа нет ничего нового. Объектный тип `serial_t` в нашей модели имеет собственный конструктор, свою версию функции проверки контрольной цифры и способ возврата информации о себе при вызове метода `print`.

Помимо конструктора и методов экземпляров, Oracle поддерживает еще две разновидности методов:

- **Статические методы** — функции и процедуры, работа которых не зависит от экземпляра объекта. Такие методы являются аналогами обычных процедур и функций PL/SQL.
- **Методы сравнения (MAP и ORDER)** — методы, определяющие критерии соответствия или сортировки. Это специальные методы экземпляров, позволяющие определить процедуру сравнения двух объектов данного типа, например при проверке на эквивалентность в PL/SQL или при сортировке объектов в SQL.

На объекты распространяется правило Oracle, гласящее, что значение неинициализированной переменной автоматически устанавливается равным NULL (более точный термин для объектов — атомарный NULL; за дополнительной информацией обращайтесь к главе 13). Как и в случае с коллекциями, в этом случае значение атрибутам объекта присваивать нельзя. Рассмотрим пример:

```

DECLARE
    mybook book_t;          -- объект объявлен, но не инициализирован
BEGIN
    IF mybook IS NULL       -- проверка вернет TRUE
    THEN
        mybook.title := 'Learning Oracle PL/SQL'; -- в этой строке выдается...
    END IF;
EXCEPTION
    WHEN ACCESS_INTO_NULL  -- ...стандартное исключение
    THEN
        ...
END;

```

Прежде чем присваивать значения атрибутам, вы обязаны инициализировать объект (создать экземпляр) одним из трех способов: с помощью метода-конструктора, путем непосредственного присваивания экземпляра другого объекта или посредством выборки из базы данных (см. раздел «Запись, выборка и использование объектов»).

Вызов методов супертипа в Oracle11g

Одно из ограничений объектно-ориентированных средств Oracle, которое было снято в Oracle11g, заключалось в возможности вызова метода супертипа, переопределенного в текущем подтипе (или подтипе более высокого уровня). До выхода Oracle11g, если метод супертипа переопределялся в подтипе, вызвать метод супертипа в экземпляре подтипа было невозможно. Сейчас такая возможность появилась.

Предположим, мы создали корневой тип для обработки и отображения информации о еде:

```

/* Файл в Сети: 11g_gen_invoc.sql */
CREATE TYPE food_t AS OBJECT (
    NAME          VARCHAR2 (100),
    food_group     VARCHAR2 (100),
    grown_in       VARCHAR2 (100),
    MEMBER FUNCTION to_string RETURN VARCHAR2
)
NOT FINAL;
/
CREATE OR REPLACE TYPE BODY food_t
IS
    MEMBER FUNCTION to_string RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'FOOD! ' || self.name || ' - '
            || self.food_group || ' - ' || self.grown_in;
    END;
END;
/

```

Далее создается подтип для представления десертов, переопределяющий метод `to_string`. При выводе информации о десерте нам хотелось бы вывести как информацию, относящуюся только к десертам, так и более общие атрибуты, но по возможности обойтись без копирования/вставки кода — лучше повторно использовать имеющийся код из типа `food_t`. До выхода Oracle11g это было невозможно, однако благодаря новому механизму вызова (`SELF AS супертун`) подтип теперь можно определить в следующем виде:

```

CREATE TYPE dessert_t UNDER food_t (
    contains_chocolate CHAR (1)
    , year_created NUMBER (4)
    , OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
);
/

CREATE OR REPLACE TYPE BODY dessert_t
IS
    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
    IS
    BEGIN
        /* Включение строки супертун (food_t) в строку подтипа.... */
        RETURN 'DESSERT! With Chocolate? '
            || contains_chocolate
            || ' created in '
            || SELF.year_created
            || chr(10)
            || (SELF as food_t).to_string;
    END;
END;
/

```

Теперь при выводе представления десертов в формате `to_string` также выводятся общие атрибуты:

```

DECLARE
    TYPE foodstuffs_nt IS TABLE OF food_t;
    fridge_contents foodstuffs_nt
        := foodstuffs_nt (
            food_t ('Eggs benedict', 'PROTEIN', 'Farm')
            , dessert_t ('Strawberries and cream'
                , 'FRUIT', 'Backyard', 'N', 2001)
        );
BEGIN
    FOR indx in 1 .. fridge_contents.COUNT
    LOOP

```

```

        DBMS_OUTPUT.put_line (RPAD ('=', 60, '='));
        DBMS_OUTPUT.put_line (fridge_contents (indx).to_string);
    END LOOP;
END;
/

```

Результат:

```

=====
FOOD! Eggs benedict - PROTEIN - Farm
=====
DESSERT! With Chocolate? N created in 2001
FOOD! Strawberries and cream - FRUIT - Backyard

```

В отличие от некоторых других объектно-ориентированных языков, в Oracle при вызове метода супертипа используется не обобщенное ключевое слово `SUPERTYPE`, а конкретный супертип из иерархии. Такое решение обладает большей гибкостью (вы можете указать нужный супертип), однако оно также означает, что имя супертипа жестко фиксируется в коде подтипа.

Запись, выборка и использование объектов

До сих пор мы рассматривали определение типов данных и создание экземпляров объектов в памяти, выделяемой для выполняющихся программ. Однако это только небольшой фрагмент «объектной картины» — конечно, Oracle позволяет сохранять объекты в базе данных.

Библиотечный каталог из нашего примера можно сохранить в базе данных как минимум двумя способами: либо в виде одной большой таблицы объектов, соответствующих элементам каталога, либо в виде набора меньших таблиц, представляющих подтипы элементов каталога. Первый способ реализуется так:

```

CREATE TABLE catalog_items OF catalog_item_t
    (CONSTRAINT catalog_items_pk PRIMARY KEY (id));

```

Эта команда предписывает Oracle создать таблицу объектов `catalog_items`, каждая строка которой представляет собой объект типа `catalog_item_t`. Объектная таблица обычно содержит по одному столбцу на атрибут:

```

SQL > DESC catalog_items

```

| Name | Null? | Type |
|-------|----------|----------------|
| ----- | ----- | ----- |
| ID | NOT NULL | NUMBER(38) |
| TITLE | | VARCHAR2(4000) |

Однако помните, что объектный тип `catalog_item_t` не допускает создания экземпляров и каждая строка таблицы на самом деле будет объектом одного из его подтипов, например книгой или периодическим изданием. Куда же попадают дополнительные атрибуты? Рассмотрим следующие допустимые команды¹:

```

INSERT INTO catalog_items
    VALUES (NEW book_t(10003, 'Perelandra', '0-684-82382-9', 222));
INSERT INTO catalog_items
    VALUES (NEW serial_t(10004, 'Time', '0040-781X', '0'));

```

В таблице необходимо выделить место для хранения значений атрибутов каждого из возможных подтипов. Чтобы выделить данную операцию, в таблицу добавляются скрытые столбцы, по одному на каждый уникальный атрибут подтипа. С точки зрения

¹ Я бы предпочел использовать именованные параметры в этих статических функциях, но эта возможность не поддерживалась до Oracle Database 11g, где появилась поддержка именованной записи для любых пользовательских функций PL/SQL, вызываемых из команд SQL.

программирования объектов это удобно, поскольку позволяет сохранить абстракцию элементов каталога.

Секция **CONSTRAINT** указывает, что столбец `id` является первичным ключом. Да, у объектных таблиц тоже могут быть первичные ключи. Кроме того, при отсутствии предложения **CONSTRAINT** Oracle автоматически сгенерирует идентификатор объекта (OID).

ЦЕПОЧКИ ВЫЗОВА МЕТОДОВ

Объект, определение типа которого выглядит следующим образом:

```
CREATE OR REPLACE TYPE chaindemo_t AS OBJECT (  
  x NUMBER, y VARCHAR2(10), z DATE,  
  MEMBER FUNCTION setx (x IN NUMBER) RETURN chaindemo_t,  
  MEMBER FUNCTION sety (y IN VARCHAR2) RETURN chaindemo_t,  
  MEMBER FUNCTION setz (z IN DATE) RETURN chaindemo_t);
```

предоставляет возможность «сцепления» вызова своих методов. Например:

```
DECLARE  
  c chaindemo_t := chaindemo_t(NULL, NULL, NULL);  
BEGIN  
  c := c.setx(1).sety('foo').setz(sysdate); -- цепочка вызовов
```

Эта команда в действительности эквивалентна следующей серии:

```
c := c.setx(1);  
c := c.sety('foo');  
c := c.setz(sysdate);
```

Каждая функция возвращает типизованный объект, который становится входным значением для следующей функции в цепочке. Реализация одного из таких методов выглядит так (другие имеют похожую реализацию):

```
MEMBER FUNCTION setx (x IN NUMBER) RETURN chaindemo_t IS  
  l_self chaindemo_t := SELF;  
BEGIN  
  l_self.x := x;  
  RETURN l_self;  
END;
```

Несколько правил, относящихся к цепочкам вызовов:

- Возвращаемое значение функции не может использоваться как параметр IN OUT следующей функции в цепочке. Функции возвращают значения, доступные только для чтения.
- Методы вызываются справа налево.
- Возвращаемое значение сцепленного метода должно относиться к объектному типу, который ожидает получить метод справа.
- Цепочка вызовов может содержать не более одной процедуры.
- Если ваша цепочка вызовов включает процедуру, она должна вызываться в крайней правой позиции в цепочке.

Идентификация объектов

В реляционных базах данных каждая строка имеет уникальный идентификатор. Объектно-ориентированные системы также обычно присваивают своим объектам уникальные идентификаторы. Программист, использующий объектно-ориентированные возможности базы данных, может объединять эти два способа идентификации. В следующей таблице перечислены основные программные конструкции, в которых могут быть задействованы идентификаторы объектов.

| Что и где | Используются ли идентификаторы объектов |
|--|--|
| Объект строки в объектной таблице | Да |
| Объектный столбец в любой таблице (или при выборке в программе PL/SQL) | Нет; используйте первичный ключ строки |
| Временный объект, создаваемый в программе PL/SQL | Нет; используйте весь объект |
| Объект строки, выбираемый из объектной таблицы в программе PL/SQL | Да, но доступен в программе только при явной выборке REF (см. далее «Использование REF») |

Пример таблицы, которая может содержать объектные столбцы:

```
CREATE TABLE my_writing_projects (
  project_id INTEGER NOT NULL PRIMARY KEY,
  start_date DATE,
  working_title VARCHAR2(4000),
  catalog_item catalog_item_t -- "объектный столбец"
);
```

Oracle Corporation считает, что объектный столбец зависит от первичного ключа строки, и он не должен идентифицироваться независимо¹.

Для любой объектной таблицы идентификатор любого объекта может быть создан на основе одного из следующих элементов:

- **Первичный ключ** — в конец команды CREATE TABLE добавляется секция OBJECT IDENTIFIER IS PRIMARY KEY.
- **Значение, генерируемое системой**, — Oracle добавляет в таблицу скрытый столбец с именем SYS_NC_OID\$ и помещает в него уникальное 16-байтовое значение типа RAW.

Идентификаторы на основе первичных ключей обычно занимают меньше места, чем системно-генерируемые, но последние обладают определенными преимуществами. Более подробное обсуждение достоинств и недостатков двух указанных методов вы найдете в документации *Oracle Application Developer's Guide — Object-Relational Features*. Пока же достаточно знать, что системно-генерируемые идентификаторы обладают следующими качествами:

- **Непрозрачность**. Хотя программы могут использовать их неявно, значение такого идентификатора обычно невозможно получить.
- **Потенциальная глобальная уникальность**. Пространство OID имеет объем, достаточный для хранения 2^{128} объектов, и теоретически позволяет идентифицировать объекты в распределенных базах данных без явных ссылок на базы данных.
- **Неизменяемость**. В данном контексте это означает, что идентификаторы нельзя обновлять. Даже после экспорта и импорта объекта его OID остается тем же, в отличие от ROWID. Для изменения OID нужно удалить объект и создать его заново.

Функция VALUE

Для извлечения объектов из базы данных Oracle предоставляет SQL-функцию VALUE. Она получает единственный аргумент, в котором передается указанный в условии FROM псевдоним таблицы, и возвращает объект того типа, на основе которого определена данная таблица. В команде SELECT это выглядит следующим образом:

```
SELECT VALUE(c)
  FROM catalog_items c;
```

¹ Противоположного мнения придерживаются эксперты в области реляционных баз данных, которые считают, что идентификаторы OID не должны использоваться для идентификации строк и только объекты столбцов должны иметь OID. См. статью Хью Дарвена (Hugh Darwen) и С. Дейта (C. J. Date) «The Third Manifesto», SIGMOD Record 24, no. 1 (март 1995 г.).

Функция `VALUE` возвращает вызывающей программе последовательность битов, а не текстовое представление значений столбцов. Однако в `SQL*Plus` существует встроенная возможность вывода объектов, поэтому результаты приведенного запроса выводятся следующим образом:

```
VALUE(C)(ID, TITLE)
-----
BOOK_T(10003, 'Perelandra', '0-684-82382-9', 222)
SERIAL_T(10004, 'Time', '0040-781X', '0')
```

В PL/SQL также имеются средства для работы с извлеченными из базы данных объектами. Сначала объявляется объектная переменная соответствующего типа:

```
DECLARE
    catalog_item catalog_item_t;
    CURSOR ccur IS
        SELECT VALUE(c)
            FROM catalog_items c;
BEGIN
    OPEN ccur;
    FETCH ccur INTO catalog_item;
    DBMS_OUTPUT.PUT_LINE('I fetched item #' || catalog_item.id);
    CLOSE ccur;
END;
```

В аргументе `PUT_LINE` используется синтаксис *переменная.атрибут*, а возвращает она значение указанного атрибута:

```
I fetched item #10003
```

Команда `FETCH` присваивает объект локальной переменной `catalog_item`, которая объявлена как переменная базового объектного типа. Это естественно, поскольку мы не знаем заранее, объект какого подтипа будет выбран из базы данных.

Заодно этот код показывает (на примере вывода значения атрибута `catalog_item.id`), что мы имеем прямой доступ к атрибутам базового типа.

Также можно использовать обычные атрибуты курсоров. Например, приведенный выше анонимный блок можно переписать таким образом:

```
DECLARE
    CURSOR ccur IS
        SELECT VALUE(c) obj
            FROM catalog_items c;
    arec ccur%ROWTYPE;
BEGIN
    OPEN ccur;
    FETCH ccur INTO arec;
    DBMS_OUTPUT.PUT_LINE('I fetched item #' || arec.obj.id);
    CLOSE ccur;
END;
```

Если нам потребуется вывести все атрибуты объекта, можно использовать метод `print`. Это вполне допустимо, поскольку он объявлен в объектном типе корневого уровня и реализован в подтипах. На этапе выполнения Oracle найдет для данного метода перегруженную версию из подтипа.

Функция `VALUE` поддерживает точечный синтаксис, обеспечивающий доступ к атрибутам и методам, но только к тем из них, которые определены в базовом типе. Например, команда

```
SELECT VALUE(c).id, VALUE(c).print()
    FROM catalog_items c;
```

возвращает результат:

```
VALUE(c).ID VALUE(c).PRINT()
```

```
-----
10003 id=10003; title=Perelandra; isbn=0-684-82382-9; pages=222
10004 id=10004; title=Time; issn=0040-781X; open_or_closed=Open
```

При работе в клиентской среде, не поддерживающей объектов Oracle, можно воспользоваться именно этой функцией.

А если попытаться прочитать только те атрибуты, которые являются уникальными для конкретного подтипа? Выполните такую команду:

```
SELECT VALUE(c).issn /* Ошибка. Атрибуты подтипа недоступны */
FROM catalog_items c;
```

В ответ Oracle выдает сообщение об ошибке. Дело в том, что объект родительского типа не предоставляет прямого доступа к атрибутам подтипа. Можно попробовать объявить переменную `book` типа `book_t` и присвоить ей объект данного подтипа в надежде, что она покажет скрытые атрибуты:

```
book := catalog_item; /* Ошибка; Oracle не выполняет предполагаемое
                        понижающее преобразование типа */
```

На этот раз я получаю ошибку PLS-00382 (неправильный тип выражения). Что происходит? Неочевидный ответ на этот вопрос приведен в следующем разделе.

Несколько завершающих замечаний по поводу выполнения DML-операций в объектных реляционных таблицах:

- Для объектной таблицы, созданной на основе объектного типа, не имеющего подтипов, возможны выборка, вставка, обновление и удаление значений всех столбцов с использованием обычных команд SQL. Таким образом, объектно-ориентированные и реляционные программы могут совместно использовать одни и те же данные.
- Традиционные команды DML не имеют доступа к скрытым столбцам, представляющим атрибуты подтипов. Для работы с такими столбцами используется «объектный DML».
- Для обновления всего объекта в таблице базы данных из программы PL/SQL можно использовать объектную команду DML, которая обновит все атрибуты (столбцы), включая уникальные для подтипа:

```
UPDATE catalog_items c SET c = object_variable WHERE ...
```

- Единственным известным способом обновления заданного столбца, уникального для подтипа, является обновление всего объекта. Например, чтобы установить количество страниц книги с идентификатором 10007 равным 1000, потребуется такая команда:

```
UPDATE catalog_items c
SET c = NEW book_t(c.id, c.title, c.publication_date, c.subject_refs,
    (SELECT TREAT(VALUE(y) AS book_t).isbn
     FROM catalog_items y
     WHERE id = 10007),
    1000)
WHERE id = 10007;
```

А теперь вернемся к проблеме, о которой я упоминал выше.

Функция TREAT

Допустим, в объявлении типа переменной PL/SQL указано имя супертипа, а ее значение принадлежит к подтипу. Как получить доступ к атрибутам и методам, специфическим для этого подтипа? Допустим, мы хотим интерпретировать элемент каталога как книгу. Эта операция называется *понижающим преобразованием* (downcasting), и иногда компилятор не может выполнить ее автоматически. В таких случаях приходится использовать функцию Oracle `TREAT`:


```

DECLARE
    book book_t;
    catalog_item catalog_item_t := NEW book_t();
BEGIN
    book := TREAT(catalog_item AS book_t);  /* Работает в 9i R2 и выше */
END;

```

или с помощью SQL (до выхода Oracle9i Release 2 функция TREAT в PL/SQL не поддерживалась):

```

DECLARE
    book book_t;
    catalog_item catalog_item_t := book_t(NULL, NULL, NULL, NULL);
BEGIN
    SELECT TREAT (catalog_item AS book_t)
        INTO book
        FROM DUAL;
END;

```

Общий синтаксис функции TREAT выглядит так:

```
TREAT (экземпляр_объекта AS подтип) [ . { атрибут | метод( аргументы...) } ]
```

Здесь *экземпляр_объекта* — это значение конкретного супертипа в объектной иерархии, а *подтип* — имя подтипа в этой иерархии. Если вы попытаетесь преобразовать объект к типу, принадлежащему другой объектной иерархии, компилятор выдаст ошибку. Когда функции TREAT передается объект из правильной иерархии, она возвращает либо преобразованный объект, либо NULL, но не ошибку.

Как и при использовании функции VALUE, при обращении к функции TREAT можно задать атрибут или метод преобразованного объекта с использованием точечного синтаксиса: DBMS_OUTPUT.PUT_LINE(TREAT (VALUE(c) AS serial_t).issn);

Если вам понадобится перебрать все объекты в таблице и выполнить определенные операции с учетом их типов, это можно сделать так:

```

DECLARE
    CURSOR ccur IS
        SELECT VALUE(c) item FROM catalog_items c;
    arec ccur%ROWTYPE;
BEGIN
    FOR arec IN ccur
    LOOP
        CASE
            WHEN arec.item IS OF (book_t)
            THEN
                DBMS_OUTPUT.PUT_LINE('Found a book with ISBN '
                    || TREAT(arec.item AS book_t).isbn);
            WHEN arec.item IS OF (serial_t)
            THEN
                DBMS_OUTPUT.PUT_LINE('Found a serial with ISSN '
                    || TREAT(arec.item AS serial_t).issn);
            ELSE
                DBMS_OUTPUT.PUT_LINE('Found unknown catalog item');
        END CASE;
    END LOOP;
END;

```

Данный блок демонстрирует применение предиката IS OF для проверки объектного типа. Синтаксис предиката выглядит многообещающе:

```
объект IS OF ( [ ONLY ] имя_типа )
```

Однако на самом деле возможности предиката ограничены — он работает только с объектными типами данных Oracle и не работает с базовыми, такими как NUMBER или DATE. Компилятор выдаст сообщение об ошибке, если при использовании предиката тип объекта не принадлежит той же иерархии, что и *тип_имя_типа*.

Обратите внимание на ключевое слово **ONLY**. По умолчанию (то есть без **ONLY**) предикат возвращает **TRUE**, если объект относится к заданному типу *или одному из его подтипов*. Если ключевое слово **ONLY** присутствует, предикат **IS OF** не сравнивает подтипы и возвращает **TRUE** только в случае точного совпадения типов.



На уровне синтаксиса вывод любого выражения **TREAT** всегда должен использоваться как функция, даже если **TREAT** вызывается для активизации процедуры. Например, если тип `book_t` содержит процедуру `set_isbn`, можно предположить, что строка следующего вида допустима:

```
TREAT(item AS book_t).set_isbn('0140714154'); -- неверно
```

Но компилятор выдает странную ошибку **PLS-00363** (выражение '**SYS_TREAT**' не может использоваться для присваивания).

Вместо этого необходимо сохранить результат во временной переменной, а затем вызвать процедуру:

```
book := TREAT(item AS book_t);
book.set_isbn('0140714154');
```

Предикат **IS OF**, как и функция **TREAT**, может использоваться в **SQL Oracle9i**, но в **PL/SQL** до **Oracle9i Release 2** он не поддерживается. Чтобы обойти это ограничение, в **Release 1** можно определить в дереве типов один или несколько дополнительных методов и, пользуясь динамической диспетчеризацией, выполнить нужную операцию на соответствующем уровне иерархии. Выбор подходящего метода понижающего преобразования зависит не только от номера версии, но и от выполняемых приложением операций.

А теперь я хочу перейти к другой интересной области: полезным возможностям, которые **Oracle** вам предоставляет при (неизбежном!) изменении архитектуры приложения.

Эволюция и создание типов

По сравнению с **Oracle8i** в **Oracle9i** достигнут огромный прогресс в области эволюции типов. Теперь **Oracle** позволяет вносить в объектные типы различные изменения, причем это возможно даже тогда, когда таблицы уже созданы и заполнены объектами.

Ранее в этой главе мы на скорую руку определили объектный тип `catalog_item_t`. Однако любой библиотекарь скажет, что для каждой книги или периодического издания желательно хранить в каталоге дату издания. Поэтому мы делаем следующее:

```
ALTER TYPE catalog_item_t
  ADD ATTRIBUTE publication_date VARCHAR2(400)
  CASCADE INCLUDING TABLE DATA;
```

Oracle распространяет изменение на соответствующие таблицы, автоматически выполняя все необходимые модификации. В конец списка атрибутов супертипа добавляется новый атрибут, а за последним столбцом супертипа в соответствующую объектную таблицу добавляется новый столбец. Если теперь выполнить команду **DESCRIBE**, которая выводит описание заданного объекта, результат будет таким:

```
SQL> DESC catalog_item_t
catalog_item_t is NOT FINAL
catalog_item_t is NOT INSTANTIABLE
```

| Name | Null? | Type |
|------------------|-------|----------------|
| ----- | ----- | ----- |
| ID | | NUMBER(38) |
| TITLE | | VARCHAR2(4000) |
| PUBLICATION_DATE | | VARCHAR2(400) |

```
METHOD
```

```

-----
MEMBER FUNCTION CK_DIGIT_OKAY RETURNS BOOLEAN
CK_DIGIT_OKAY IS NOT INSTANTIABLE

```

```

METHOD

```

```

-----
MEMBER FUNCTION PRINT RETURNS VARCHAR2

```

Для таблицы эта команда выдаст следующий результат:

```
SQL> DESC catalog_items
```

| Name | Null? | Type |
|------------------|----------|----------------|
| ----- | ----- | ----- |
| ID | NOT NULL | NUMBER(38) |
| TITLE | | VARCHAR2(4000) |
| PUBLICATION_DATE | | VARCHAR2(400) |

Фактически команда ALTER TYPE изменяет почти все, но, к сожалению, она не настолько умна, чтобы переписать наши методы. Особенно важны конструкторы, поскольку придется изменить их сигнатуры. Проблема решается удалением и повторным добавлением метода.

Для удаления метода из спецификации book_t необходимо выполнить такую команду:

```

ALTER TYPE book_t
  DROP CONSTRUCTOR FUNCTION book_t (id INTEGER DEFAULT NULL,
    title VARCHAR2 DEFAULT NULL,
    isbn VARCHAR2 DEFAULT NULL,
    pages INTEGER DEFAULT NULL)
  RETURN SELF AS RESULT
  CASCADE;

```

Обратите внимание на полную спецификацию функции. Она гарантирует, что удаляется именно тот метод, который нужен, поскольку могут существовать его перегруженные версии. (Стоит заметить, что значения по умолчанию указывать не обязательно.)

Соответствующая операция добавления метода выглядит так же просто:

```

ALTER TYPE book_t
  ADD CONSTRUCTOR FUNCTION book_t (id INTEGER DEFAULT NULL,
    title VARCHAR2 DEFAULT NULL,
    publication_date VARCHAR2 DEFAULT NULL,
    isbn VARCHAR2 DEFAULT NULL,
    pages INTEGER DEFAULT NULL)
  RETURN SELF AS RESULT
  CASCADE;

```

Далее нужно аналогичным образом модифицировать тип serial_t, а затем заменить два соответствующих тела типов командой CREATE OR REPLACE TYPE BODY.

Кроме того, следует просмотреть все методы на предмет необходимости изменений (например, дату издания желательно включить в метод print).

Кстати говоря, типы можно удалять, для этого используется следующая команда:

```
DROP TYPE имя_типа [ FORCE ];
```

Ключевым словом FORCE (доступным только в Oracle 11g Release 2 и выше) следует пользоваться осторожно, поскольку отменить последствия выполнения команды уже не удастся. Все объектные типы и объектные таблицы, зависящие от типа, становятся неприменимыми. Если в таблицах имеются столбцы данного типа, Oracle сделает их недоступными. Для удаления подтипа, который используется в определениях таблиц, можно задать команду DROP TYPE в такой форме:

```
DROP TYPE имя_подтипа VALIDATE;
```

С ключевым словом VALIDATE Oracle просматривает таблицы и удаляет подтип только в случае, если таблицы не содержат ни одного экземпляра этого подтипа. Таким образом можно избежать разрушительных последствий от применения ключевого слова FORCE.

И снова указатели

Среди объектно-ориентированных функций Oracle появилась возможность хранения значений типа REF. Они представляют собой логические указатели на конкретную строку объектной таблицы. В ссылке хранится следующая информация:

- первичный ключ строки или системно-генерируемый идентификатор объекта;
- уникальный идентификатор таблицы;
- информация о физическом местоположении строки на диске в виде ее значения типа ROWID.

Значение REF представляет собой длинную шестнадцатеричную строку:

```
SQL> SELECT REF(c) FROM catalog_items c WHERE ROWNUM = 1;
REF(C)
```

```
-----
00002802099FC431FBE5F20599E0340003BA0F1F139FC431FBE5F10599E0340003BA0F1F130240000
```

Запросы и программы могут использовать значения REF для выборки строк объектной таблицы без указания ее имени. Давайте посмотрим, как REF-ссылки могут применяться в библиотечном каталоге.

Использование REF-ссылок

Библиотечный фонд обычно классифицируется по тематике. Например, книга, которую вы сейчас читаете, может относиться к трем категориям:

- Oracle (Компьютеры);
- PL/SQL (Языки программирования);
- реляционные базы данных.

Для классификации используется древовидная структура: категория «Компьютеры» является родительской для Oracle, а категория «Языки программирования» — родительской для PL/SQL.

Категории и хранящиеся в библиотеке объекты связаны отношением «многие-ко-многим»: книга может относиться к нескольким категориям, а к категории может принадлежать множество книг. В нашем библиотечном каталоге книги всех категорий размещены в одной таблице. В реляционной методологии создается промежуточная таблица, разделяющая отношение «многие-ко-многим» на два отношения «один-ко-многим». Однако объектно-реляционный подход предлагает другое решение.

Начнем с создания объектного типа для категории:

```
CREATE TYPE subject_t AS OBJECT (
    name VARCHAR2(2000),
    broader_term_ref REF subject_t
);
```

Для каждой категории в таблице хранится ее имя и имя вышестоящей категории. Чтобы не размещать один объект в другом, мы будем хранить только ссылку на вышестоящую категорию. Поэтому в третьей строке определения атрибут `broader_term_ref` задается как REF-ссылка на объект того же типа. Далее создается таблица категорий.

О внешнем ключе таблицы стоит упомянуть особо. Он указывает на таблицу с реляционным первичным ключом, но имеет тип REF, поэтому Oracle знает, что в качестве внешнего ключа нужно использовать идентификаторы объектов. Поддержка ограничений внешнего ключа типа REF демонстрирует связь между объектным и реляционным мирами в Oracle.

Несколько команд вставки данных в таблицу (с использованием системного конструктора):

```

INSERT INTO subjects VALUES (subject_t('Computer file', NULL));
INSERT INTO subjects VALUES (subject_t('Computer program language', NULL));
INSERT INTO subjects VALUES (subject_t('Relational databases', NULL));
INSERT INTO subjects VALUES (subject_t('Oracle',
    (SELECT REF(s) FROM subjects s WHERE name = 'Computer file')));
INSERT INTO subjects VALUES (subject_t('PL/SQL',
    (SELECT REF(s) FROM subjects s WHERE name = 'Computer program language')));

```

Теперь можно вывести содержимое таблицы subjects:

```
SQL> SELECT VALUE(s) FROM subjects s;
```

```

VALUE(S)(NAME, BROADER_TERM_REF)
-----
SUBJECT_T('Computer file', NULL)
SUBJECT_T('Computer program language', NULL)
SUBJECT_T('Oracle', 00002202089FC431FBE6FC0599E0340003BA0F1F139FC431FBE6690599E03
40003BA0F1F13)
SUBJECT_T('PL/SQL', 00002202089FC431FBE6FC0599E0340003BA0F1F139FC431FBE6690599E03
40003BA0F1F13)
SUBJECT_T('Relational databases', NULL)

```

Интересно, но есть ли от этого какая-нибудь практическая польза? Оказывается, есть: Oracle умеет автоматически «разрешать» такие ссылки. Например, с помощью функции Deref можно перейти к строке, на которую указывает ссылка:

```

SELECT s.name, Deref(s.broader_term_ref).name bt
FROM subjects s;

```

Разрешение похоже на внешнее автоматическое объединение. Иначе говоря, если ссылка равна NULL или недействительна, строка включается в результирующий набор, но значение целевого объекта (и столбца) будет равно NULL.

В Oracle для функции Deref введено удобное сокращение — точечный синтаксис определения атрибута, извлекаемого из целевого объекта:

```
SELECT s.name, s.broader_term_ref.name bt FROM subjects s;
```

Оба запроса выводят следующий результат:

| NAME | BT |
|---------------------------|---------------------------|
| Computer file | |
| Computer program language | |
| Oracle | Computer file |
| PL/SQL | Computer program language |
| Relational databases | |

Обратите внимание на то, что в обоих формах необходим псевдоним таблицы:

```

SELECT псевдоним_таблицы.столбец_ref
FROM имя_таблицы псевдоним_таблицы

```

Ссылки могут использоваться и в секции WHERE. Например, вывести все подкатегории из категории «Computer program language» можно с помощью следующего запроса:

```

SELECT VALUE(s).name FROM subjects s
WHERE s.broader_term_ref.name = 'Computer program language';

```

Хотя в нашем примере в таблице используется ссылка на эту же таблицу, на практике ссылки могут указывать на любую объектную таблицу базы данных. Чтобы продемонстрировать сказанное, вернемся к определению базового типа catalog_item_t и добавим в него атрибут, предназначенный для хранения коллекции ссылок. Каждый элемент каталога будет связан с набором категорий. Сначала создадим коллекцию ссылок на категории:

```
CREATE TYPE subject_refs_t AS TABLE OF REF subject_t;
```

Теперь каждый элемент каталога можно ассоциировать с любым количеством категорий:

```
ALTER TYPE catalog_item_t
  ADD ATTRIBUTE subject_refs subject_refs_t
  CASCADE INCLUDING TABLE DATA;
```

Не будем останавливаться на модификации соответствующих методов зависимых типов и перейдем к добавлению строк в каталог. Это делается с помощью следующей команды SQL:

```
INSERT INTO catalog_items
VALUES (NEW book_t(10007,
  'Oracle PL/SQL Programming',
  'Sept 1997',
  CAST(MULTISET(SELECT REF(s)
    FROM subjects s
    WHERE name IN ('Oracle', 'PL/SQL', 'Relational databases'))
  AS subject_refs_t),
  '1-56592-335-9',
  987));
```

Конструкция CAST/MULTISET на ходу преобразует REF-ссылки на темы в коллекцию (см. раздел «Работа с коллекциями» главы 12).

Вот как выглядит чуть более понятный эквивалентный фрагмент PL/SQL:

```
DECLARE
  subrefs subject_refs_t;
BEGIN
  SELECT REF(s)
  BULK COLLECT INTO subrefs
  FROM subjects s
  WHERE name IN ('Oracle', 'PL/SQL', 'Relational databases');
  INSERT INTO catalog_items VALUES (NEW book_t(10007,
    'Oracle PL/SQL Programming', 'Sept 1997', subrefs, '1-56592-335-9', 987));
END;
```

Проще говоря, мы получаем в этом коде REF-ссылки на три категории и сохраняем их для конкретной книги.

Навигация с использованием REF-ссылок настолько удобна, что мы рассмотрим еще один пример:

```
SELECT VALUE(s).name
|| ' (' || VALUE(s).broader_term_ref.name || ')' plsql_subjects
FROM TABLE(SELECT subject_refs
  FROM catalog_items
  WHERE id=10007) s;
```

Код извлекает данные из таблицы `subjects`, при этом она даже не упоминается по имени. (Функция `TABLE` преобразует коллекцию в виртуальную таблицу.) Результаты выполнения приведенного запроса имеют следующий вид:

```
PLSQL_SUBJECTS
-----
Relational databases ()
PL/SQL (Computer program language)
Oracle (Computer file)
```

Что дает эта технология программисту, кроме автоматического перехода по ссылкам в SQL? К сожалению, почти ничего. Дело в том, что ссылки имеют *сильную типизацию* — иначе говоря, столбец типа REF может указывать только на объект того типа, который указан в его определении. Внешние же ключи могут указывать на любые строки, для которых определен первичный ключ или уникальный индекс.

Пакет UTL_REF

Встроенный пакет UTL_REF выполняет операцию разыменования (получения значения объекта, на который указывает ссылка) без явного вызова SQL. Это позволяет приложению программным путем блокировать, извлекать, обновлять и удалять объекты по REF-ссылкам. Чтобы рассмотреть эту возможность, мы добавим в объектный тип `subject_t` еще один метод:

```
MEMBER FUNCTION print_bt (str IN VARCHAR2)
    RETURN VARCHAR2
IS
    bt subject_t;
BEGIN
    IF SELF.broader_term_ref IS NULL
    THEN
        RETURN str;
    ELSE
        UTL_REF.SELECT_OBJECT(SELF.broader_term_ref, bt);
        RETURN bt.print_bt(NVL(str,SELF.name)) || ' (' || bt.name || ')';
    END IF;
END;
```

Эта рекурсивная процедура просматривает иерархию от текущей категории до самой верхней — родительской.

УЛУЧШЕННАЯ ПОДДЕРЖКА REF-ССЫЛОК В С

В OCI (Oracle Call Interface), интерфейсе Oracle для языков C/C++, и даже в Pro*C, объектные ссылки приносят больше пользы. Помимо навигации по REF-ссылкам (по аналогии с PL/SQL), OCI предоставляет механизм комплексного получения объектов COR (Complex Object Retrieval). COR позволяет получить объект и всех его «соседей», на которые указывают REF-ссылки, за один вызов. OCI и Pro*C поддерживают кэш объектов на стороне клиента, который позволяет приложению загружать объекты в память клиента и выполнять с ними операции (выборка, вставка, обновление, слияние, удаление) так, как если бы они находились в базе данных. Затем приложение одним вызовом передает все изменения на сервер. Помимо расширения функциональности, этот механизм сокращает количество пересылок данных и способствует повышению общего быстродействия. С другой стороны, создание кэша данных Oracle за пределами сервера создает ряд проблем, связанных с конкуренцией и блокировками.

При использовании процедур из пакета UTL_REF аргумент REF должен иметь тот же тип, что и объектный аргумент. Ниже приведен полный список процедур этого пакета:

- **UTL_REF.SELECT_OBJECT** (*объектная_ссылка* IN, *объектная_переменная* OUT);
Находит объект, на который указывает аргумент *объектная_ссылка*, и извлекает его копию в *объектную_переменную*.
- **UTL_REF.SELECT_OBJECT_WITH_CR** (*объектная_ссылка* IN, *объектная_переменная* OUT);
Является аналогом **SELECT_OBJECT**, но делает копию («снимок») объекта. Данная версия позволяет избежать ошибки ORA-4091, которая может произойти при обновлении объектной таблицы и присваивании значения функции, использующей пакет UTL_REF для разыменования ссылки объекта из обновляемой таблицы.
- **UTL_REF.LOCK_OBJECT** (*объектная_ссылка* IN);
Устанавливает блокировку объекта, на который указывает *объектная_ссылка*, но не извлекает его.

- `UTL_REF.LOCK_OBJECT (объектная_ссылка IN, объектная_переменная OUT);`
Устанавливает блокировку объекта, на который указывает *объектная_ссылка*, и извлекает его копию в объектную переменную.
- `UTL_REF.UPDATE_OBJECT (объектная_ссылка IN, объектная_переменная IN);`
Заменяет объект, на который указывает *объектная_ссылка*, значением *объектной_переменной*. Операция обновляет все столбцы соответствующей объектной таблицы.
- `UTL_REF.DELETE_OBJECT (объектная_ссылка IN);`
Удаляет объект, на который указывает *объектная_ссылка*.

Объектные ссылки и иерархии типов

Все перечисленные подпрограммы являются процедурами, а не функциями¹, а их параметры имеют квазислабую типизацию. Другими словами, базе данных не нужно знать во время компиляции точный тип данных, если REF-ссылки соответствуют объектным переменным.

Рассмотрим особенности объектных ссылок, которые проявляются при работе с иерархиями типов. Предположим, в программе объявлены следующие переменные:

```
DECLARE
    book book_t;
    item catalog_item_t;
    itemref REF catalog_item_t;
    bookref REF book_t;
```

Вы уже видели, что при присваивании извлекаемого по ссылке значения сильнотипизированной переменной все работает нормально:

```
SELECT REF(c) INTO itemref
FROM catalog_items c WHERE id = 10007;
```

Подобным образом можно извлечь объект по ссылке с использованием процедуры `SELECT_OBJECT`:

```
UTL_REF.select_object(itemref, item);
```

или:

```
SELECT Deref(itemref) INTO item FROM DUAL;
```

Однако непосредственное понижающее преобразование объектного типа при выполнении этой операции не допускается:

```
SELECT REF(c)
    INTO bookref /* Ошибка */
FROM catalog_items c WHERE id = 10007;
```

С этой целью можно воспользоваться функцией `TREAT`, которая поддерживает работу со ссылками:

```
SELECT TREAT(REF(c) AS REF book_t)
    INTO bookref
FROM catalog_items c WHERE id = 10007;
```

Привести объект к родительскому типу можно как явно:

```
UTL_REF.select_object(TREAT(bookref AS ref catalog_item_t), item);
```

так и неявно:

```
SELECT Deref(bookref) INTO item FROM DUAL;
```

¹ Что довольно странно — как минимум `SELECT_OBJECT` было бы логичнее оформить в виде функции.

И хотя непосредственно понизить объект до дочернего типа при использовании функции Deref нельзя:

```
SELECT Deref(itemref)
  INTO book    /* Ошибка */
  FROM DUAL;
```

функция Treat и здесь придет на помощь:

```
SELECT Deref(Treat(itemref AS Ref book_t))
  INTO book
  FROM catalog_items c WHERE id = 10007;
```

Неявное понижающее преобразование также можно выполнить с помощью процедуры из пакета UTL_REF:

```
UTL_REF.select_object(itemref, book);
```

Висячие ссылки

Ссылка Ref может не указывать ни на один объект; тогда она называется *висячей*. Так может случиться, когда объект, на который указывает хранящаяся в базе данных ссылка, удален. Такая аномальная ситуация может возникнуть при отсутствии внешнего ключа, препятствующего подобному удалению.

Для поиска висячих ссылок используется оператор IS Dangling:

```
SELECT VALUE(s) FROM subjects s
WHERE broader_term_ref IS Dangling;
```

Перейдем к средствам Oracle, предназначенным для работы с данными неизвестных или изменяющихся типов.

Типы данных ANY

В главе 13 упоминалось о том, что Oracle поддерживает тип ANYDATA, который может содержать данные других встроенных или пользовательских типов. Пользуясь типом ANYDATA, подпрограмма PL/SQL может, например, сохранять в базе данных, считывать и обрабатывать элементы данных любого из типов SQL, и вам не нужно создавать десятки ее перегруженных версий. Эта возможность была предназначена для создания очередей: приложение помещает «нечто» в очередь, при этом очереди не нужно знать, к какому типу данных относится помещенный элемент.

Для работы с произвольными типами данных Oracle предоставляет следующие типы и пакеты:

- Тип ANYDATA — инкапсулирует любые элементы данных типов SQL в самодокументированные структуры данных.
- Тип ANYTYPE — совместно с типом ANYDATA применяется для считывания описания структуры данных. Может использоваться и сам по себе для создания временных объектных типов.
- Пакет DBMS_TYPES — содержит только константы, которые упрощают интерпретацию типов данных, используемых в объекте ANYDATA.
- Тип ANYDATASET — аналог ANYDATA, но его содержимым является один или несколько экземпляров типа данных.

Чего не умеет ANYDATA

Допустим, вы решили написать функцию, которая преобразует любые данные в строковое представление. Вы начинаете со следующей спецификации:

```
FUNCTION printany (whatever IN ANYDATA) RETURN VARCHAR2;
```

рассчитывая вызывать ее следующим образом:

```
DBMS_OUTPUT.PUT_LINE(printany(SYSDATE));           -- нет
DBMS_OUTPUT.PUT_LINE(printany(NEW book_t(111)));   -- снова нет
DBMS_OUTPUT.PUT_LINE(printany('Hello world'));     -- и еще раз нет
```

К сожалению, эти вызовы не работают. ANYDATA представляет собой инкапсуляцию других типов, поэтому данные в типе ANYDATA необходимо сначала преобразовать к определенному типу одним из встроенных статических методов:

```
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertDate(SYSDATE)));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertObject(NEW book_t(12345))));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertVarchar2('Hello world')));
```

Короче говоря, ANYDATA не заменяет перегрузки данных.

Обработка данных ANYDATA

Следующая программа возвращает строковую версию содержимого любой переменной. Представленная версия работает только с числами, строками, датами, объектами и ссылками, но ее можно расширить для поддержки практически любых типов данных.

```
/* Файл в Сети: printany.fun */
1  FUNCTION printany (adata IN ANYDATA)
2      RETURN VARCHAR2
3  AS
4      aType ANYTYPE;
5      retval VARCHAR2(32767);
6      result_code PLS_INTEGER;
7  BEGIN
8      CASE adata.GetType(aType)
9      WHEN DBMS_TYPES.TYPECODE_NUMBER THEN
10         RETURN 'NUMBER: ' || TO_CHAR(adata.AccessNumber);
11      WHEN DBMS_TYPES.TYPECODE_VARCHAR2 THEN
12         RETURN 'VARCHAR2: ' || adata.AccessVarchar2;
13      WHEN DBMS_TYPES.TYPECODE_CHAR THEN
14         RETURN 'CHAR: ' || RTRIM(adata.AccessChar);
15      WHEN DBMS_TYPES.TYPECODE_DATE THEN
16         RETURN 'DATE: ' || TO_CHAR(adata.AccessDate,
17                                     'YYYY-MM-DD hh24:mi:ss');
18      WHEN DBMS_TYPES.TYPECODE_OBJECT THEN
19         EXECUTE IMMEDIATE 'DECLARE ' ||
20             ' myobj ' || adata.GetTypeName || ' '; ' ||
21             ' myad anydata := :ad; ' ||
22             'BEGIN ' ||
23             ' :res := myad.GetObject(myobj); ' ||
24             ' :ret := myobj.print(); ' ||
25             'END;'
26         USING IN adata, OUT result_code, OUT retval;
27         retval := adata.GetTypeName || ': ' || retval;
28      WHEN DBMS_TYPES.TYPECODE_REF THEN
29         EXECUTE IMMEDIATE 'DECLARE ' ||
30             ' myref ' || adata.GetTypeName || ' '; ' ||
31             ' myobj ' || SUBSTR(adata.GetTypeName,
32                                INSTR(adata.GetTypeName, ' ')) || ' '; '
33             ' myad anydata := :ad; ' ||
34             'BEGIN ' ||
35             ' :res := myad.GetREF(myref); ' ||
36             ' UTL_REF.SELECT_OBJECT(myref, myobj); ' ||
37             ' :ret := myobj.print(); ' ||
38             'END;'
39         USING IN adata, OUT result_code, OUT retval;
40         retval := adata.GetTypeName || ': ' || retval;
```

```

41     ELSE
42         retval := '<data of type ' || adata.GetTypeName || '>';
43     END CASE;
44
45     RETURN retval;
46
47 EXCEPTION
48     WHEN OTHERS
49     THEN
50         IF INSTR(SQLERRM, 'component ''PRINT'' must be declared') > 0
51         THEN
52             RETURN adata.GetTypeName || ': <no print() function>';
53         ELSE
54             RETURN 'Error: ' || SQLERRM;
55         END IF;
56 END;
```

Рассмотрим код более подробно.

| Строки | Описание |
|-----------------------|--|
| 5 | Если для хранения результата понадобится временная переменная, мы резервируем для нее 32 Кбайт. Поскольку для больших переменных типа VARCHAR2 PL память выделяется динамически, зарезервированная память будет использована только в случае необходимости |
| 6 | Значение переменной result_code (см. строки 26 и 39) не используется в данном примере, но его требует API ANYDATA |
| 8 | Тип ANYDATA включает метод GetType, возвращающий код соответствующего типа данных. Его спецификация: MEMBER FUNCTION ANYDATA.GetType (OUT NOCOPY ANYTYPE) RETURN typecode_integer; Но для использования этого метода необходимо объявить переменную типа ANYTYPE, в значении которой Oracle хранит подробную информацию об инкапсулированном типе |
| 9, 11, 13, 15, 18, 28 | В этих выражениях используются константы из встроенного пакета DBMS_TYPES |
| 10, 12, 14, 16 | В этих операторах используются функции ANYDATA.AccessNNN, введенные в Oracle9i Release 2. В Release 1 можно было использовать аналогичные функции GetNNN, но для этого требовалась временная локальная переменная |
| 19–26 | Чтобы вывести информацию об объекте без многочисленных обращений к словарию данных, динамический анонимный блок создает объект нужного типа и вызывает его метод print() |
| 29–39 | Задача этого фрагмента — найти объект по указателю и вернуть его содержимое. Он будет работать только при наличии у объекта метода print() |
| 45–52 | При попытке вывести информацию об объекте, не имеющем метода print, генерируется ошибка времени выполнения. В этой части кода она перехватывается и обрабатывается |

Выполнение приведенных ранее вызовов:

```

DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertDate(SYSDATE)));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertObject(NEW book_t(12345)));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertVarchar2('Hello world')));
```

дает следующий результат:

```

DATE: 2005-03-10 16:00:25
SCOTT.BOOK_T: id=12345; title=; publication_date=; isbn=; pages=
VARCHAR2: Hello world
```

Как видите, тип данных ANYDATA не так удобен, как иерархии наследования, поскольку он требует явного преобразования данных. Однако он позволяет создавать столбцы таблиц и атрибуты объектов, в которых можно хранить данные практически любого типа¹.

¹ На момент написания книги в таблице не могли храниться значения ANYDATA, инкапсулирующие расширяемый объект или объект, входящий в состав иерархии типов.

Создание временного типа данных

PL/SQL не поддерживает определения новых объектных типов в разделе объявлений программы, но с помощью встроенных типов `ANY` можно создавать «временные» типы, существующие только во время выполнения программы. Значения типа, созданного с помощью `ANYTYPE`, можно передавать в качестве параметра. Кроме того, можно создавать его экземпляры как значения типа `ANYDATA`. Рассмотрим следующий код:

```
/* Создание (анонимного) временного типа с двумя атрибутами: number, date */
FUNCTION create_a_type
RETURN ANYTYPE
AS
    mytype ANYTYPE;
BEGIN
    ANYTYPE.BeginCreate(typecode => DBMS_TYPES.TYPECODE_OBJECT,
                        atype => mytype);
    mytype.AddAttr(typecode => DBMS_TYPES.TYPECODE_NUMBER,
                  aname => 'just_a_number',
                  prec => 38,
                  scale => 0,
                  len => NULL,
                  csid => NULL,
                  csfrm => NULL);
    mytype.AddAttr(typecode => DBMS_TYPES.TYPECODE_DATE,
                  aname => 'just_a_date',
                  prec => 5,
                  scale => 5,
                  len => NULL,
                  csid => NULL,
                  csfrm => NULL);
    mytype.EndCreate;
    RETURN mytype;
END;
```

Процесс состоит из трех основных шагов:

1. Создание типа начинается с вызова статической процедуры `BeginCreate`. Она возвращает инициализированный объект `ANYTYPE`.
2. С помощью процедуры `AddAttr` последовательно добавляются атрибуты.
3. Вызывается процедура `EndCreate`.

Аналогичным образом перед использованием типа присваиваются значения его атрибутам:

```
DECLARE
    ltype ANYTYPE := create_a_type;
    l_any ANYDATA;
BEGIN
    ANYDATA.BeginCreate(dtype => ltype, adata => l_any);
    l_any.SetNumber(num => 12345);
    l_any.SetDate(dat => SYSDATE);
    l_any.EndCreate;
END;
```

Если структура данных заранее не известна, информацию о ней можно получить с помощью методов `ANYTYPE` (в частности, `GetAttrElemInfo`) в сочетании с методами `ANYDATA`. `Get`. (Пример приведен в файле `anyObject.sql` на сайте книги.)

Сделай сам

Некоторые приверженцы объектно-ориентированного программирования считают, что каждый объектный тип должен быть самодостаточным. Если объект предназначен

для хранения в базе данных, он должен «уметь» сохранять себя и содержать методы обновления, удаления и выборки. При таком подходе наш тип следовало бы дополнить такими методами:

```
ALTER TYPE catalog_item_t
  ADD MEMBER PROCEDURE remove
  CASCADE;
TYPE BODY catalog_item_t
AS
  ...
  MEMBER PROCEDURE remove
  IS
  BEGIN
    DELETE catalog_items
    WHERE id = SELF.id;
    SELF := NULL;
  END;
END;
```

(Кстати говоря, методы-деструкторы Oracle не поддерживает.) Определяя метод на уровне супертипа, мы автоматически включаем его и во все подтипы. В данном случае предполагается, что все соответствующие объекты находятся в одной таблице, однако в некоторых приложениях может потребоваться дополнительная логика поиска объектов. (Кроме того, в реальном приложении версия этого метода может включать код для выполнения дополнительных операций, таких как удаление зависимых объектов и/или архивирование данных перед окончательным удалением объекта.)

Если предположить, что перед записью временного объекта на диск приложение должно его модифицировать в памяти, вставку и обновление можно объединить в одном методе, который мы назовем *save*:

```
ALTER TYPE catalog_item_t
  ADD MEMBER PROCEDURE save,
  CASCADE;
TYPE BODY catalog_item_t
AS
  ...
  MEMBER PROCEDURE save
  IS
  BEGIN
    UPDATE catalog_items c
    SET c = SELF
    WHERE id = SELF.id;
    IF SQL%ROWCOUNT = 0
    THEN
      INSERT INTO catalog_items VALUES (SELF);
    END IF;
  END;
END;
```

Процедура заменяет значения всех столбцов, даже если они не изменились. В результате этой операции могут активизироваться «лишние» триггеры, а также выполняться ненужные операции ввода/вывода. К сожалению, это одно из неизбежных следствий объектной методологии. При более тщательном программировании можно было бы избежать модификации неизменившихся столбцов супертипа, но столбцы подтипа ни в какой поддерживаемой Oracle разновидности команды *UPDATE* в отдельности недоступны.

Из всех операций над объектными таблицами сложнее всего инкапсулировать выборку, что связано с разнообразием возможных условий в предложении *WHERE* и большим количеством форм данных результата. Как следствие, спецификация критерия запроса может значительно усложниться. Что касается результатов запроса, то и они могут быть представлены в одном из следующих видов:

- коллекция объектов;
- коллекция REF-ссылок;
- результирующий набор с конвейерной организацией;
- курсорная переменная (с сильной или слабой типизацией).

На выбор представления могут повлиять требования к приложению и необходимая для него программная среда. В усеченном примере ниже используется курсорная переменная:

```
ALTER TYPE catalog_item_t
  ADD STATIC FUNCTION cursor_for_query (typename IN VARCHAR2 DEFAULT NULL,
    title IN VARCHAR2 DEFAULT NULL,
    att1 IN VARCHAR2 DEFAULT NULL,
    val1 IN VARCHAR2 DEFAULT NULL)
    RETURN SYS_REFCURSOR
  CASCADE;
```

Мы используем статический метод, который возвращает встроенный тип SYS_REFCURSOR (слаботипизированный курсор, который Oracle предоставляет для удобства). Это позволяет клиентской программе организовать перебор результатов. Параметры att1 и val1 предоставляют средства получения информации о парах «атрибут/значение», специфических для подтипа; в реальной версии этой программы стоило бы передавать коллекцию таких пар.

А теперь рассмотрим пример выполнения запроса:

```
DECLARE
  catalog_item catalog_item_t;
  l_refcur SYS_REFCURSOR;
BEGIN
  l_refcur := catalog_item_t.cursor_for_query(
    typename => 'book_t',
    title => 'Oracle PL/SQL Programming');
  LOOP
    FETCH l_refcur INTO catalog_item;
    EXIT WHEN l_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Matching item:' || catalog_item.print);
  END LOOP;
  CLOSE l_refcur;
END;
```

Результат выглядит так:

```
Matching item:id=10007; title=Oracle PL/SQL Programming;
  publication_date=Sept 1997;
 isbn=1-56592-335-9; pages=987
```

Реализация:

```
1      MEMBER PROCEDURE save
2      IS
3      BEGIN
4          UPDATE catalog_items c
5              SET c = SELF
6              WHERE id = SELF.id;
7              IF SQL%ROWCOUNT = 0
8              THEN
9                  INSERT INTO catalog_items VALUES (SELF);
10             END IF;
11     END;
12
13     STATIC FUNCTION cursor_for_query (typename IN VARCHAR2 DEFAULT NULL,
14         title IN VARCHAR2 DEFAULT NULL,
15         att1 IN VARCHAR2 DEFAULT NULL,
16         val1 IN VARCHAR2 DEFAULT NULL)
17         RETURN SYS_REFCURSOR
```

```

18      IS
19      l_sqlstr VARCHAR2(1024);
20      l_refcur SYS_REFCURSOR;
21      BEGIN
22      l_sqlstr := 'SELECT VALUE(c) FROM catalog_items c WHERE 1=1 ';
23      IF title IS NOT NULL
24      THEN
25      l_sqlstr := l_sqlstr || 'AND title = :t ';
26      END IF;
27
28      IF typename IS NOT NULL
29      THEN
30      IF att1 IS NOT NULL
31      THEN
32      l_sqlstr := l_sqlstr
33      || 'AND TREAT(SELF AS '
34      || typename || ').' || att1 || ' ';
35      IF val1 IS NULL
36      THEN
37      l_sqlstr := l_sqlstr || 'IS NULL ';
38      ELSE
39      l_sqlstr := l_sqlstr || '=:v1 ';
40      END IF;
41      END IF;
42      l_sqlstr := l_sqlstr || 'AND VALUE(c) IS OF
43      (' || typename || ') ';
44      END IF;
45
46      l_sqlstr := 'BEGIN OPEN :lcur FOR ' || l_sqlstr || '; END;';
47
48      IF title IS NULL AND att1 IS NULL
49      THEN
50      EXECUTE IMMEDIATE l_sqlstr USING IN OUT l_refcur;
51      ELSIF title IS NOT NULL AND att1 IS NULL
52      THEN
53      EXECUTE IMMEDIATE l_sqlstr USING IN OUT l_refcur, IN title;
54      ELSIF title IS NOT NULL AND att1 IS NOT NULL
55      THEN
56      EXECUTE IMMEDIATE l_sqlstr
57      USING IN OUT l_refcur, IN title, IN att1;
58      END IF;
59
60      RETURN l_refcur;
61      END;

```

Разобраться в динамическом SQL не так просто, поэтому я приведу функцию, которая будет сгенерирована для предыдущего запроса:

```

BEGIN
  OPEN :lcur FOR
    SELECT VALUE(c)
    FROM catalog_items c
    WHERE 1=1
      AND title = :t
      AND VALUE(c) IS OF (book_t);
END;

```

Этот подход хорош еще и тем, что вам не приходится изменять код запросам при добавлении нового подтипа в дерево наследования.

Сравнение объектов

До сих пор в наших примерах использовались объектные таблицы, то есть таблицы, в которых каждая строка представляет собой объект, построенный командой CREATE

TABLE...OF. При работе с ними доступны такие специальные возможности, как переходы по ссылкам и интерпретация целого объекта (а не отдельных его столбцов) как единицы ввода/вывода.

Объектный тип также может использоваться в качестве типа отдельного столбца таблицы (в терминологии Oracle используется название «объектные столбцы»). Допустим, мы хотим завести журнал изменений для таблицы `catalog_items` и фиксировать в нем все операции вставки, обновления и удаления строк этой таблицы.

```
CREATE TABLE catalog_history (
  id INTEGER NOT NULL PRIMARY KEY,
  action CHAR(1) NOT NULL,
  action_time TIMESTAMP DEFAULT (SYSTIMESTAMP) NOT NULL,
  old_item catalog_item_t,
  new_item catalog_item_t)
NESTED TABLE old_item.subject_refs STORE AS catalog_history_old_subrefs
NESTED TABLE new_item.subject_refs STORE AS catalog_history_new_subrefs;
```

ПСЕВДОСТОЛБЕЦ OBJECT_VALUE

У любознательного читателя может возникнуть вопрос — как именно можно заполнить такую таблицу, как `catalog_history`, включающую объектные столбцы с типом, имеющим подтипы? Хочется надеяться, что это можно сделать в триггере уровня таблицы. Остается понять, как сохранить значения атрибутов всех подтипов. На помощь приходит псевдостолбец. Возьмем следующий фрагмент:

```
TRIGGER catalog_hist_upd_trg
AFTER UPDATE ON catalog_items
FOR EACH ROW
BEGIN
  INSERT INTO catalog_history (id,
    action,
    action_time,
    old_item,
    new_item)
  VALUES (catalog_history_seq.NEXTVAL,
    'U',
    SYSTIMESTAMP,
    :OLD.OBJECT_VALUE,
    :NEW.OBJECT_VALUE);
END;
```

Oracle предоставляет доступ к подтипам со всеми атрибутами через псевдостолбец `OBJECT_VALUE`. Однако такой способ работает только в Oracle Database 10g и выше; правда, похожий псевдостолбец `SYS_NC_ROWINFO$` доступен и в предыдущих версиях, но я обнаружил, что в этом конкретном приложении он не работает.

Псевдостолбец `OBJECT_VALUE` также используется для других целей, не связанных с подтипами; например, он может пригодиться при создании объектных представлений с использованием секции `WITH OBJECT IDENTIFIER`.

Но при заполнении таблицы объектами возникает вопрос — как будет действовать Oracle, если мы потребуем отсортировать или проиндексировать таблицу по одному из столбцов типа `catalog_item_t`? Существуют четыре метода сравнения объектов:

- Сравнение на уровне атрибутов. При сортировке, создании индексов и сравнении задаются соответствующие атрибуты.
- Стандартный способ SQL. Oracle умеет выполнять простейшую проверку равенства. Два объекта считаются равными, если они определены на основе одного и того же

типа, а все их соответствующие атрибуты равны. Это сравнение может быть выполнено, когда объекты имеют только скалярные атрибуты (не коллекции и не LOB) и для них не определен метод `MAP` или `ORDER`.

- Метод `MAP`. Можно определить специальную функцию-метод, возвращающую не значение объекта, а соответствующее ему значение одного из типов данных, которые Oracle умеет сравнивать. Метод работает только при отсутствии метода `ORDER`.
- Метод `ORDER`. Другая специальная функция, которая сравнивает два объекта и возвращает флаг, обозначающий их относительный порядок. Работает только при отсутствии метода `MAP`.

От стандартного метода сравнения SQL пользы немного, поэтому мы не будем его рассматривать. О других, более полезных способах сравнения объектов рассказывается в следующем разделе.

Сравнение на уровне атрибутов

Сравнение на уровне атрибутов может быть не совсем тем, что требуется, но зато оно легко реализуется в PL/SQL и даже в SQL, если только вы не забудете задать псевдоним таблицы в команде SQL. Oracle позволяет ссылаться на атрибуты с помощью точечного синтаксиса:

```
SELECT * FROM catalog_history c
WHERE c.old_item.id > 10000
ORDER BY NVL(TREAT(c.old_item AS book_t).isbn, TREAT
(c.old_item AS serial_t).issn)
```

Создание индекса на основе атрибутов выполняется так же просто:

```
CREATE INDEX catalog_history_old_id_idx ON catalog_history c (c.old_item.id);
```

Метод MAP

Методы `MAP` и `ORDER` позволяют выполнять команды следующего вида:

```
SELECT * FROM catalog_history
ORDER BY old_item;
```

```
IF old_item > new_item
THEN ...
```

Сначала рассмотрим метод `MAP`. Его простейший вариант добавляется в `catalog_item_t` следующим образом:

```
ALTER TYPE catalog_item_t
ADD MAP MEMBER FUNCTION mapit RETURN NUMBER
CASCADE;
```

```
TYPE BODY catalog_item_t
AS ...
MAP MEMBER FUNCTION mapit RETURN NUMBER
IS
BEGIN
RETURN id;
END;
...
END;
```

Если предположить, что сортировка по идентификаторам имеет смысл, то теперь мы можем сортировать и сравнивать элементы каталога, причем Oracle будет автоматически вызывать этот метод при необходимости. Функция не обязательно должна быть такой простой; например, она может представлять собой скалярное значение, вычисляемое на основе всех атрибутов типа, объединенных способом, который имеет для библиотекарей определенный смысл.

У метода `MAP` есть один побочный эффект: проверка равенства может определяться неподходящим способом. При наличии такого метода два объекта считаются равными при равенстве возвращаемых значений. Когда вам нужно сравнивать объекты поочередным анализом их атрибутов, создайте собственный метод (не `MAP`) или же используйте метод `ORDER`.

Метод `ORDER`

Альтернативой методу `MAP` является функция `ORDER`, которая сравнивает два объекта: `SELF` и один объект того же типа, заданный в качестве ее аргумента. Эта функция должна возвращать целочисленное значение, которое в зависимости от относительного порядка двух объектов может быть положительным, нулевым или отрицательным (табл. 26.2).

Таблица 26.2. Поведение функции `ORDER`

| Порядок элементов | Значение функции <code>ORDER</code> |
|----------------------------------|---|
| <code>SELF < аргумент</code> | Любое отрицательное число (обычно <code>-1</code>) |
| <code>SELF = аргумент</code> | <code>0</code> |
| <code>SELF > аргумент</code> | Любое положительное число (обычно <code>1</code>) |
| Результат сравнения не определен | <code>NULL</code> |

Рассмотрим пример нетривиального использования метода `ORDER`:

```

1  ALTER TYPE catalog_item_t
2      DROP MAP MEMBER FUNCTION mapit RETURN NUMBER
3      CASCADE;
4
5  ALTER TYPE catalog_item_t
6      ADD ORDER MEMBER FUNCTION orderit (obj2 IN catalog_item_t)
7          RETURN INTEGER
8      CASCADE;
9
10 TYPE BODY catalog_item_t
11 AS ...
12     ORDER MEMBER FUNCTION orderit (obj2 IN catalog_item_t)
13         RETURN INTEGER
14     IS
15         self_gt_o2 CONSTANT PLS_INTEGER := 1;
16         eq CONSTANT PLS_INTEGER := 0;
17         o2_gt_self CONSTANT PLS_INTEGER := -1;
18         l_matching_count NUMBER;
19     BEGIN
20         CASE
21             WHEN obj2 IS OF (book_t) AND SELF IS OF (serial_t) THEN
22                 RETURN o2_gt_self;
23             WHEN obj2 IS OF (serial_t) AND SELF IS OF (book_t) THEN
24                 RETURN self_gt_o2;
25             ELSE
26                 IF obj2.title = SELF.title
27                     AND obj2.publication_date = SELF.publication_date
28                 THEN
29                     IF obj2.subject_refs IS NOT NULL
30                         AND SELF.subject_refs IS NOT NULL
31                         AND obj2.subject_refs.COUNT = SELF.subject_refs.COUNT
32                     THEN
33                         SELECT COUNT(*) INTO l_matching_count FROM
34                             (SELECT *
35                              FROM TABLE(SELECT CAST(SELF.subject_refs AS subject_
36 refs_t)
37                                     FROM dual)
38                             INTERSECT

```

```

38          SELECT *
39          FROM TABLE(SELECT CAST(obj2.subject_refs AS subject_
refs_t)
40                          FROM dual));
41          IF l_matching_count = SELF.subject_refs.COUNT
42          THEN
43              RETURN eq;
44          END IF;
45          END IF;
46          END IF;
47          RETURN NULL;
48      END CASE;
49  END;
50  ...
51  END;

```

Важнейшие моменты в этом коде:

| Строки | Описание |
|--------|--|
| 21–24 | Означает, что при сортировке книги будут размещаться перед периодикой |
| 26–46 | Проверка равенства. Поскольку Oracle не умеет сравнивать коллекции, мы воспользуемся способностью использовать выборку из коллекции, как из таблицы. Проверив, содержит ли реляционное пересечение двух коллекций ожидаемое количество элементов, можно определить, имеет ли каждый элемент первой коллекции равный элемент во второй коллекции (предполагается именно такой критерий равенства) |

Этот метод приведен исключительно в учебных целях. Вряд ли он подойдет для реального приложения, поскольку не проверяет атрибуты, специфические для подтипа.

Рекомендации по проведению сравнения

В заключение приведем несколько правил и рекомендаций, касающихся методов сравнения объектов.

- Методы **MAP** и **ORDER** не могут сосуществовать в одном объектном типе; используйте только один из них.
- При сортировке или сравнении большого количества объектов, как это бывает в командах **SQL**, Oracle рекомендует применять метод **MAP**. Дело в том, что внутренняя оптимизация сокращает количество вызовов, тогда как метод **ORDER** должен обязательно вызываться для каждого сравнения.
- Oracle игнорирует имена методов; называйте их как угодно.
- Подтипы могут включать метод **MAP**, но только если он есть у супертипа.
- Подтипы не могут включать метод **ORDER**; логика сравнения должна определяться в супертипе.

Объектные представления

Объектные расширения Oracle предоставляют программистам PL/SQL богатые возможности для разработки новых систем, однако маловероятно, что ради них вы захотите полностью перепроектировать старые системы. Чтобы существующие приложения могли использовать преимущества новых объектных функций, в Oracle реализованы объектные представления, обладающие рядом уникальных преимуществ.

- **Объектное представление удаленных данных.** В Oracle9i еще не поддерживаются объектные таблицы и физические REF-ссылки в распределенных базах данных, но можно создавать объектные представления и виртуальные REF-ссылки, представляющие удаленные данные реляционных баз как объекты.
- **Виртуальная денормализация.** В реляционной или даже объектно-реляционной базе данных обычно реализуются однонаправленные отношения. Например, книга

относится к определенному количеству категорий. Объектные представления позволяют установить обратное соответствие; в частности объект-категория может включать коллекцию REF-ссылок, указывающих на все книги этой категории.

- **Эффективность доступа к объектам.** В приложениях OCI (Oracle Call Interface) могут использоваться программные конструкции, обеспечивающие удобное извлечение, кэширование и обновление объектных данных. Сокращая количество обращений приложения к серверу базы данных, они повышают производительность и, кроме того, делают код более лаконичным.
- **Большая гибкость в отношении изменения объектной модели.** Хотя в новых версиях Oracle имеются прекрасные возможности в области эволюции типов, добавление и удаление атрибутов объектов по-прежнему сопряжено с перемещением таблицы по диску, чего администраторы баз данных очень не любят. Перекомпиляция объектных представлений не имеет таких последствий.

В то же время у объектных представлений есть определенные недостатки.

- **Производительность.** Объектные представления обрабатываются не так быстро, как этого хотелось бы.
- **Виртуальные ссылки.** Виртуальные ссылки не хранятся в базе данных, а создаются в ходе выполнения программы. Это может вызвать определенные проблемы, если однажды вы вдруг захотите преобразовать объектные представления в объектные таблицы.

В Oracle также имеются другие средства, расширяющие возможности любых представлений, включая и объектные. Среди них стоит выделить коллекции и триггеры **INSTEAD OF**.

- **Коллекции.** Если взять две реляционные таблицы, связанные отношением «главная — подчиненная», можно создать для них представление, возвращающее строки подчиненной таблицы в виде одного нескаларного атрибута (коллекции) строки главной таблицы.
- **Триггеры INSTEAD OF.** С помощью триггера **INSTEAD OF** можно указать Oracle, как именно должны выполняться операции вставки, обновления и удаления данных в представлениях.

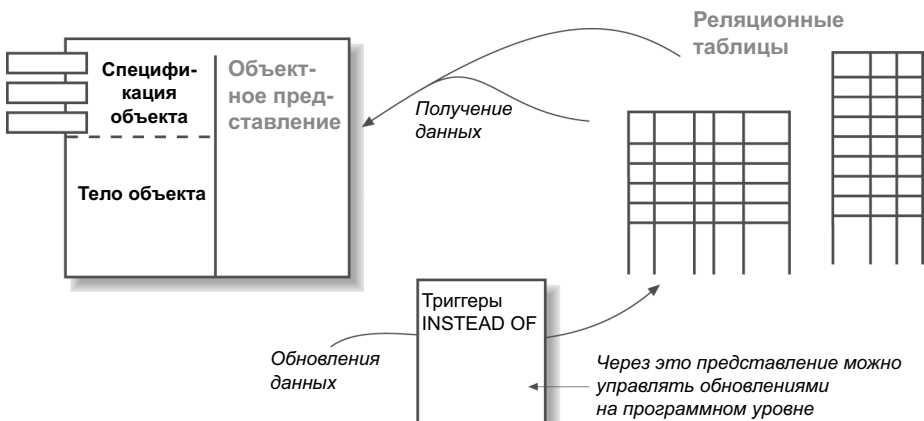


Рис. 26.2. Объектные представления позволяют связать определения объектных типов с (существующими) реляционными таблицами

С точки зрения объектной теории объектные представления имеют один второстепенный недостаток — они не обеспечивают инкапсуляции. Поскольку приложение применяет команды **INSERT**, **UPDATE**, **MERGE** и **DELETE** непосредственно к содержимому реляционных баз

данных, это отменяет все преимущества инкапсуляции, обычно присущие объектному подходу. Но поскольку Oracle все равно не поддерживает ни частных атрибутов, ни частных методов, потеря невелика.

Если вы решили наложить объектные представления поверх существующей системы, может оказаться, что новые приложения будут пользоваться дополнительными преимуществами, а унаследованные системы будут работать так же, как прежде. Применение объектных представлений проиллюстрировано на рис. 26.2.

В следующих разделах рассматриваются особенности объектных представлений, в том числе различия между объектными представлениями и объектными таблицами, которые представляют особый интерес для программистов PL/SQL.

Пример реляционной системы

Во втором крупном примере этой главы мы рассмотрим применение объектных представлений в приложении базы данных компании, занимающейся графическим дизайном. Реляционное приложение используется для работы с информацией об используемых изображениях (GIF, JPEG и т. д.). Изображения хранятся в файлах, но информация о них содержится в таблицах базы данных. Чтобы художнику легче было искать нужные изображения, с каждым из них связывается одно или несколько ключевых слов. Данные об изображениях и ключевые слова хранятся в двух таблицах, связанных отношением «главная — подчиненная».

В базе данных имеется таблица поставщиков изображений:

```
CREATE TABLE suppliers (  
    id INTEGER NOT NULL PRIMARY KEY,  
    name VARCHAR2(400) NOT NULL  
);
```

Также имеется таблица метаданных изображений:

```
CREATE TABLE images (  
    image_id INTEGER NOT NULL PRIMARY KEY,  
    file_name VARCHAR2(512) NOT NULL,  
    file_type VARCHAR2(12) NOT NULL,  
    supplier_id INTEGER REFERENCES suppliers (id),  
    supplier_rights_descriptor VARCHAR2(256),  
    bytes INTEGER  
);
```

Однако не все изображения поступают от поставщиков; если идентификатор поставщика (`supplier_id`) равен NULL, значит, изображение создано работниками самой компании.

В базе данных имеется еще одна таблица, которая содержит ключевые слова, связанные с изображениями:

```
CREATE TABLE keywords (  
    image_id INTEGER NOT NULL REFERENCES images (image_id),  
    keyword VARCHAR2(45) NOT NULL,  
    CONSTRAINT keywords_pk PRIMARY KEY (image_id, keyword)  
);
```

Предположим, таблицы заполнены следующими данными:

```
INSERT INTO suppliers VALUES (101, 'Joe's Graphics');  
INSERT INTO suppliers VALUES (102, 'Image Bar and Grill');  
INSERT INTO images VALUES (100001, '/files/web/60s/smiley_face.png', 'image/png',  
    101, 'fair use', 813);  
INSERT INTO images VALUES (100002, '/files/web/60s/peace_symbol.gif', 'image/  
gif',  
    101, 'fair use', 972);
```

```

INSERT INTO images VALUES (100003, '/files/web/00s/towers.jpg',
    'image/jpeg', NULL,
    NULL, 2104);
INSERT INTO KEYWORDS VALUES (100001, 'SIXTIES');
INSERT INTO KEYWORDS VALUES (100001, 'HAPPY FACE');
INSERT INTO KEYWORDS VALUES (100002, 'SIXTIES');
INSERT INTO KEYWORDS VALUES (100002, 'PEACE SYMBOL');
INSERT INTO KEYWORDS VALUES (100002, 'JERRY RUBIN');

```

В следующих разделах описываются объектные представления, определенные на основе этих данных:

- первое представление определяется на основе типа изображения, включающего в качестве атрибута коллекцию ключевых слов;
- второе представление определяется на основе подтипа из иерархии объектных типов. В него включаются характеристики изображений, полученных от поставщиков;
- последнее представление содержит ключевые слова и соответствующие обратные ссылки на изображения.

Объектное представление с атрибутом-коллекцией

Перед созданием объектного типа для первого представления нужно определить тип коллекции для хранения ключевых слов. В данном случае используем вложенную таблицу, поскольку порядок ключевых слов не важен, а их максимальное количество не ограничено¹:

```
CREATE TYPE keyword_tab_t AS TABLE OF VARCHAR2(45);
```

Объектный тип для хранения изображения определяется очень просто:

```

CREATE TYPE image_t AS OBJECT (
    image_id INTEGER,
    image_file BFILE,
    file_type VARCHAR2(12),
    bytes INTEGER,
    keywords keyword_tab_t
);

```

Если файлы изображений и сервер базы данных располагаются на одном компьютере, то вместо имени файла можно использовать тип данных **BFILE**. Нужно создать «каталог», то есть используемый только в Oracle псевдоним каталога операционной системы, в котором хранятся изображения. Зададим корневой каталог, поскольку в столбце **file_name** указаны полные пути к файлам.

```
CREATE DIRECTORY rootdir AS '/';
```



Скорее всего, вы не будете обладать привилегиями для работы с файлами в корневом каталоге; укажите тот каталог, с которым вы можете работать.

Мы пока не определили связь между реляционными таблицами и объектным типом. Это две совершенно независимые сущности, и наложение определения на таблицы происходит только при создании объектного представления:

```

CREATE VIEW images_v
    OF image_t
    WITH OBJECT IDENTIFIER (image_id)

```

¹ Если порядок важен или максимальное количество ключевых слов ограничено (относительно небольшим числом), лучше использовать коллекцию типа **VARRAY**.

```

AS
  SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
         i.file_type, i.bytes,
         CAST (MULTISET (SELECT keyword
                        FROM keywords k
                        WHERE k.image_id = i.image_id)
              AS keyword_tab_t)
  FROM images i;

```

Два компонента этой команды характерны только для объектных представлений:

- **OF image_t** — означает, что будут возвращены объекты типа **image_t**.
- **WITH OBJECT IDENTIFIER (image_id)** — чтобы возвращаемые в представлении данные были похожи на настоящие экземпляры объектов, им нужен объектный идентификатор. Определив первичный ключ как основу для виртуального OID, можно пользоваться преимуществами REF-навигации по объектам представления.

Итак, объектное представление создано, но что же с ним можно делать? Прежде всего, из него можно извлекать данные, как из объектной таблицы. Например, в SQL*Plus запрос

```
SQL> SELECT image_id, keywords FROM images_v;
```

вернет следующие данные:

```

IMAGE_ID KEYWORDS
-----
100003 KEYWORD_TAB_T()
100001 KEYWORD_TAB_T('HAPPY FACE', 'SIXTIES')
100002 KEYWORD_TAB_T('JERRY RUBIN', 'PEACE SYMBOL', 'SIXTIES')

```

Чтобы все это еще более походило на объекты, можно добавить в определение типа методы, в частности метод **print()**:

```

ALTER TYPE image_t
  ADD MEMBER FUNCTION print RETURN VARCHAR2
  CASCADE;

CREATE OR REPLACE TYPE BODY image_t
AS
  MEMBER FUNCTION print
    RETURN VARCHAR2
  IS
    filename images.file_name%TYPE;
    dirname VARCHAR2(30);
    keyword_list VARCHAR2(32767);
  BEGIN
    DBMS_LOB.FILEGETNAME(SELF.image_file, dirname, filename);
    IF SELF.keywords IS NOT NULL
    THEN
      FOR key_elt IN 1..SELF.keywords.COUNT
      LOOP
        keyword_list := keyword_list || ', ' || SELF.keywords(key_elt);
      END LOOP;
    END IF;
    RETURN 'Id=' || SELF.image_id || '; File=' || filename
           || '; keywords=' || SUBSTR(keyword_list, 3);
  END;
END;

```

Этот пример показывает, как сформировать «плоский» список ключевых слов, содержащихся в виртуальной коллекции.

Еще некоторые полезные возможности объектных представлений:

- **Использование виртуальных REF-ссылок** — указатели на виртуальные объекты, подробно рассматриваются в разделе «Различия между объектными представлениями и объектными таблицами» (см. далее).

- **Написание триггеров INSTEAD OF** с возможностью выполнения прямых операций с содержимым представления. Дополнительная информация по этой теме приведена в разделе «Триггеры INSTEAD OF» (см. далее).

NULL ИЛИ НЕТ?

NULL-коллекция — не то же самое, что инициализированная коллекция с нулем элементов. Изображение 100003 не имеет ключевых слов, но объектное представление ошибочно возвращает пустую, но инициализированную коллекцию. Чтобы получить «настоящее» значение NULL, можно воспользоваться функцией DECODE для проверки числа ключевых слов:

```
CREATE OR REPLACE VIEW images_v
  OF image_t
  WITH OBJECT IDENTIFIER (image_id)
AS
  SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
         i.file_type, i.bytes,
         DECODE((SELECT COUNT(*)
                  FROM keywords k2
                  WHERE k2.image_id = i.image_id),
                0, NULL,
                CAST (MULTISET (SELECT keyword
                                FROM keywords k
                                WHERE k.image_id = i.image_id)
                     AS keyword_tab_t))
  FROM images i;
```

Иначе говоря, при отсутствии ключевых слов возвращается NULL; в остальных случаях возвращается выражение CAST/MULTISET. С этим представлением команда SELECT...WHERE image_id=100003 дает следующий результат:

```
IMAGE_ID KEYWORDS
```

```
-----
100003
```

Пожалуй, эта концептуальная чистота не стоит лишних операций ввода/вывода (или необходимости разбираться в запутанной команде SELECT).

Объектные подпредставления

Если в нашем примере изображения определенного вида должны обрабатываться отдельно от других, можно создать для них подтип. Выделим в отдельный подтип изображения, у которых есть поставщики. В нем будет содержаться ссылка на объект поставщика, определенный следующим образом:

```
CREATE TYPE supplier_t AS OBJECT (
  id INTEGER,
  name VARCHAR2(400)
);
```

Соответствующее объектное представление выглядит так:

```
CREATE VIEW suppliers_v
  OF supplier_t
  WITH OBJECT IDENTIFIER (id)
AS
  SELECT id, name
  FROM suppliers;
```

Теперь изменим или удалим и повторно создадим базовый тип, но уже с секцией NOT FINAL:

```
ALTER TYPE image_t NOT FINAL CASCADE;
```


чтобы затем создать для него подтип:

```
CREATE TYPE supplied_images_t UNDER image_t (
    supplier_ref REF supplier_t,
    supplier_rights_descriptor VARCHAR2(256)
);
```

Завершив подготовку, мы создаем представление на основе подтипа и включаем в его определение предложение UNDER:

```
CREATE VIEW supplied_images_v
    OF supplied_images_t          UNDER images_v
AS
    SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
           i.file_type, i.bytes,
           CAST (MULTISET (SELECT keyword
                           FROM keywords k
                           WHERE k.image_id = i.image_id)
               AS keyword_tab_t),
           MAKE_REF(suppliers_v, supplier_id),
           supplier_rights_descriptor
    FROM images i
    WHERE supplier_id IS NOT NULL;
```

Oracle не позволяет при формировании подпредставления запрашивать данные из суперпредставления, поэтому в данном случае мы обращаемся непосредственно к базовой таблице, а условие WHERE ограничивает набор извлекаемых записей. Также обратите внимание на то, что в подпредставлениях не используется секция WITH OBJECT IDENTIFIER, поскольку они наследуют идентификаторы объектов от суперпредставления.

В этом запросе используется новая функция MAKE_REF, предназначенная для вычисления ссылки на виртуальный объект. В данном случае он представляет поставщика, возвращаемого представлением suppliers_v. Синтаксис функции MAKE_REF:

```
FUNCTION MAKE_REF (представление, список_значений) RETURN ссылка;
```

Здесь *представление* — это объектное представление, на которое должна указывать ссылка, заданная в значении аргумента ссылка, а *список_значений* — разделенный запятыми список значений столбцов, типы данных которых должны в точности соответствовать типам данных OID-атрибутов представления.

Учтите, что функция MAKE_REF всего лишь применяет встроенный алгоритм Oracle для построения REF-значения. Однако, как и REF-ссылки, виртуальные ссылки могут не указывать на реальные объекты.

А теперь неожиданный результат — хотя мы не изменяли суперпредставление, изображения от поставщиков теперь выводятся им дважды (то есть дублируются):

```
SQL> SELECT COUNT(*), image_id FROM images_v GROUP BY image_id;
```

| COUNT(*) | IMAGE_ID |
|----------|----------|
| 2 | 100001 |
| 2 | 100002 |
| 1 | 100003 |

Oracle возвращает логическое объединение (UNION ALL) двух запросов: к суперпредставлению и подпредставлению. И в этом есть смысл, ведь изображение от поставщика не перестает быть изображением. Для удаления дубликатов включите в родительское представление условие WHERE, которое исключает строки, возвращаемые в подпредставлении:

```
CREATE OR REPLACE VIEW images_v AS
...
WHERE supplier_id IS NULL;
```

Объектное представление с обратным отношением

Для демонстрации виртуальной денормализации можно создать объектный тип для ключевых слов и соответствующее представление, связывающее ключевые слова с изображениями, которые они описывают:

```
CREATE TYPE image_refs_t AS TABLE OF REF image_t;
```

```
CREATE TYPE keyword_t AS OBJECT (
    keyword VARCHAR2(45),
    image_refs image_refs_t);
```

Определение представления выглядит так:

```
CREATE OR REPLACE VIEW keywords_v
    OF keyword_t
    WITH OBJECT IDENTIFIER (keyword)
AS
    SELECT keyword, CAST(MULTISET(SELECT MAKE_REF(images_v, image_id)
                                FROM keywords
                                WHERE keyword = main.keyword)
                        AS image_refs_t)
    FROM (SELECT DISTINCT keyword FROM keywords) main;
```

Запросы к данному представлению не будут отличаться быстротой; ведь при этом нужно выполнять операцию `SELECT DISTINCT`, так как в базе данных нет справочной таблицы ключевых слов. Даже без использования объектных функций с точки зрения времени эти запросы обходились бы дорого.

Вы можете справедливо заметить, что использование `MAKE_REF` не обязательно; для получения `REF` также можно воспользоваться внутренним запросом к `images_v` (вместо таблицы `keywords`). В общем случае `MAKE_REF` работает быстрее, чем поиск по объектному представлению, а в некоторых ситуациях такой поиск вообще невозможен.

Как бы то ни было, на этой стадии я могу выполнять запросы типа следующего:

```
SQL> SELECT Deref(VALUE(i)).print()
       2 FROM keywords_v v, TABLE(v.image_refs) i
       3 WHERE keyword = 'SIXTIES';
```

```
Deref(VALUE(I)).PRINT()
```

```
-----
Id=100001; File=/files/web/60s/smiley_face.gif; keywords=HAPPY FACE, SIXTIES
Id=100002; File=/files/web/60s/peace_symbol.gif; keywords=JERRY RUBIN, PEACE,
SIXTIES
```

То есть я могу вывести список всех изображений, помеченных ключевым словом `SIXTIES`, наряду с другими ключевыми словами и атрибутами.

Триггеры INSTEAD OF

Синтаксис триггеров `INSTEAD OF` был описан в главе 19, и в этом разделе он не рассматривается. Мы обсудим только применение этих триггеров для обновления объектных представлений. Если вы намерены освоить объектный подход, вас может заинтересовать, не являются ли триггеры `INSTEAD OF` просто реляционным дополнением, позволяющим приложениям выполнять DML-операции. Рассмотрев аргументы «за» и «против», вы сможете решить, какой подход лучше выбрать для своего приложения.

Аргументы «против»

Для инкапсуляции DML-операций гораздо лучше триггеров `INSTEAD OF` подходят пакеты и объектные методы PL/SQL. Логика такого триггера легко переносится

в альтернативную конструкцию PL/SQL, имеющую более универсальное применение. Иначе говоря, если для выполнения DML в системе используется давно стандартизированный и отлаженный механизм пакетов и методов, то эти триггеры в него могут совершенно не вписаться и лишь усложнят вашу работу.

Более того, даже Oracle предупреждает, что триггерами не стоит злоупотреблять, потому что они могут создавать сложные взаимозависимости. Если триггер **INSTEAD OF** выполняет DML-инструкцию для таблиц, у которых имеются другие триггеры, выполняющие DML-инструкции для других таблиц с триггерами, то... как вы это все будете отлаживать?

Аргументы «за»

Значительную часть логики, обычно реализуемой в пакете или теле метода, можно перенести в триггер **INSTEAD OF**. В сочетании с продуманным набором привилегий это позволит защитить данные лучше, чем с помощью методов и пакетов.

Если вы пользуетесь таким клиентским средством, как Oracle Forms, то при создании в форме «блока» на основе представления, а не таблицы триггеры позволяют применять гораздо больше встроенных возможностей этого продукта.

При использовании OCI, когда объектное представление не подлежит модификации, а кэшируемые данные объектного представления нужно быстро отправлять обратно на сервер, триггеры **INSTEAD OF** просто *необходимы*.

Главный вопрос

Куда следует помещать команды, выполняющие вставку, обновление и удаление данных, особенно при использовании объектных представлений? Если вы хотите локализовать эти операции на сервере, имеются как минимум три варианта: пакеты PL/SQL, объектные методы и триггеры **INSTEAD OF**.

В табл. 26.3 приведены сравнительные характеристики этих вариантов. Заметьте, что мы рассматриваем их исключительно с точки зрения размещения DML-операций над объектными представлениями.

Таблица 26.3. Сравнение способов инкапсуляции DML-операций над объектными представлениями

| Характеристика | Пакет PL/SQL | Объектный метод | Триггер INSTEAD OF |
|---|--|--|---|
| Соответствие объектно-ориентированному подходу | Потенциально очень хорошее | Прекрасное | Потенциально очень хорошее |
| Возможность модификации при изменении схемы | Имеется, код легко модифицировать и независимо перекомпилировать | Имеется (в Oracle9i) | Имеется |
| Риск непредвиденных последствий | Низкий | Низкий | Высокий; триггеры могут иметь скрытые зависимости |
| Взаимодействие со стандартными функциями клиентских средств (в частности, Oracle Developer) | Программист должен добавить код для всех клиентских транзакционных триггеров | Программист должен добавить код для всех клиентских транзакционных триггеров | Хорошо реализуется для типов верхнего уровня (однако не существует серверного триггера INSTEAD OF LOCK) |
| Возможность включаться и отключаться по желанию | Отсутствует | Отсутствует | Имеется (путем отключения и включения триггера) |

Как видите, явного победителя здесь нет. В каждом варианте имеются преимущества, важность которых зависит от конкретного приложения.

У триггеров `INSTEAD OF` в иерархии представлений есть одна важная особенность: для каждого уровня иерархии необходимо создавать отдельный триггер. При выполнении команды DML применительно к подпредставлению запускается триггер подпредставления, а применительно к суперпредставлению — триггер суперпредставления.

Триггеры `INSTEAD OF` могут использоваться в сочетании с пакетами PL/SQL и/или объектными методами для обеспечения многоуровневой инкапсуляции. Например:

```
TRIGGER images_v_insert
INSTEAD OF INSERT ON images_v
FOR EACH ROW
BEGIN
    /* Вызов процедуры из пакета для выполнения вставки. */
    manage_image.create_one(:NEW.image_id, :NEW.file_type,
                           :NEW.file_name, :NEW.bytes, :NEW.keywords);
END;
```

В идеале разработчик должен заранее выбрать архитектуру и конструктивные решения, а не набивать приложению всевозможными функциями Oracle. Используйте любую функцию лишь при условии, что она действительно полезна в приложении и согласуется с принятыми общими концепциями разработки.

Различия между объектными представлениями и объектными таблицами

Кроме очевидных различий между объектными представлениями и объектной таблицей, необходимо знать и о более тонких различиях:

- уникальность OID-идентификаторов;
- возможность хранения в базе данных REF-ссылок;
- REF-ссылки на неуникальные OID-идентификаторы.

Давайте рассмотрим каждое из перечисленных различий.

Уникальность OID-идентификаторов

Объектная таблица всегда содержит уникальный идентификатор объекта (генерируемый системой или производный от первичного ключа). Технически возможно (хотя и нежелательно) создать объектную таблицу с дублирующимися строками, но экземпляры объектов в таблице все равно будут иметь уникальные идентификаторы. Данную операцию можно выполнить двумя способами:

- **Дублирование OID в одном представлении.** Объектное представление может содержать несколько экземпляров объектов (строк) с одинаковыми значениями OID. Вы уже видели пример суперпредставления с дубликатами объектов.
- **Дублирование OID в нескольких представлениях.** Если объектное представление создано на основе объектной таблицы или представления и OID определяется с помощью ключевого слова `DEFAULT`, представление содержит идентификаторы объектов, соответствующие идентификаторам базовой структуры.

Скорее всего, вам подойдет второй способ, поскольку отдельные представления — это, по сути, сохраненные результаты запросов.

Хранение в базе данных REF-ссылок

Если вы разрабатываете приложение с «физическими» объектными таблицами, REF-ссылки на объекты могут храниться в других таблицах; в конце концов, REF — это двоичное значение, которое используется Oracle в качестве указателя на объект.

Но при попытке сохранения виртуальной ссылки (то есть ссылки на строку объектного представления) в реальной таблице Oracle выдаст ошибку. Поскольку виртуальная

ссылка зависит от значений столбцов, вместо нее нужно сохранять эти значения. Это скорее неудобство, чем серьезное ограничение, и все же жаль, что объектные таблицы нельзя совмещать с объектными представлениями или преобразовывать их в объектные таблицы. Было бы хорошо иметь возможность создать объектную таблицу:

```
CREATE TABLE images2 OF image_t
  NESTED TABLE keywords STORE AS keyword_tab;
```

и затем заполнить ее данными из представления:

```
INSERT INTO images2 /* Недопустимо, поскольку images_v содержит REF */
  SELECT VALUE(i) FROM images_v i;
```

К сожалению, Oracle выдает ошибку, однако в будущем все может измениться.

Ссылки на неуникальные OID

Вряд ли возможна стабильная работа программы, содержащей ссылки на неуникальный OID при работе с объектными таблицами. Что произойдет в результате создания ссылки на объект объектного представления, содержащего несколько объектов с одинаковыми OID? Такая ситуация является аномальной; не стоит создавать представления с дублирующимися OID.

Наши эксперименты показали, что вызов функции Deref для такой виртуальной ссылки возвращает объект — вероятно, первый из найденных Oracle.

Сопровождение объектных типов и объектных представлений

При работе с объектными типами существует несколько способов получения информации о созданных вами типах и представлениях. Если возможностей команды SQL*Plus DESCRIBE окажется недостаточно, вероятно, придется перейти на прямые запросы к словарю данных Oracle.

Словарь данных

Типы, определяемые пользователем (объекты и коллекции), в словаре данных относятся к категории TYPE. Определения и тела объектных типов находятся в представлении USER_SOURCE (или DBA_SOURCE, или ALL_SOURCE), как и спецификации и тела пакетов. В табл. 26.4 перечислены некоторые полезные запросы.

Таблица 26.4. Элементы словаря данных для объектных типов

| Вопрос | Запрос, используемый для получения ответа |
|---|--|
| Какие объектные типы и типы коллекций я создал? | SELECT * FROM user_types; SELECT * FROM user_objects WHERE object_type = 'TYPE'; |
| Как выглядят мои иерархии объектных типов? | SELECT RPAD(' ', 3*(LEVEL-1)) type_name FROM user_types WHERE typecode = 'OBJECT' CONNECT BY PRIOR type_name = supertype_name; |
| Какие атрибуты содержит тип foo? | SELECT * FROM user_type_attrs WHERE type_name = 'FOO'; |
| Какие методы содержит тип foo? | SELECT * FROM user_type_methods WHERE type_name = 'FOO'; |
| Какие параметры получают методы foo? | SELECT * FROM user_method_params WHERE type_name = 'FOO'; |
| Какой тип данных возвращает метод bar типа foo? | SELECT * FROM user_method_results WHERE type_name = 'FOO' AND method_name = 'BAR'; |

продолжение ⇨

Таблица 26.4 (продолжение)

| Вопрос | Запрос, используемый для получения ответа |
|---|--|
| Как выглядит исходный код foo, включая все команды ALTER? | SELECT text FROM user_source WHERE name = 'FOO' AND type = 'TYPE' /* или 'TYPE BODY' */ ORDER BY line; |
| Какие объектные таблицы реализуют foo? | SELECT table_name FROM user_object_tables WHERE table_type = 'FOO'; |
| Какие столбцы содержит объектная таблица foo_tab (включая скрытые)? | SELECT column_name, data_type, hidden_column, virtual_column FROM user_tab_cols WHERE table_name = 'FOO_TAB'; |
| Какие столбцы реализуют foo? | SELECT table_name, column_name FROM user_tab_columns WHERE data_type = 'FOO'; |
| Какие объекты базы данных зависят от foo? | SELECT name, type FROM user_dependencies WHERE referenced_name = 'FOO'; |
| Какие объектные представления я создал, какие OID при этом использовались? | SELECT view_name, view_type, oid_text FROM user_views WHERE type_text IS NOT NULL; |
| Как выглядит моя иерархия представлений? (Требуется использования временной таблицы в версиях Oracle, которые не могут использовать подзапрос с CONNECT BY) | CREATE TABLE uvtemp AS SELECT v.view_name, v.view_type, v.superview_name, v1.view_type superview_type FROM user_views v, user_views v1 WHERE v.superview_name = v1.view_name (+); SELECT RPAD(' ', 3*(LEVEL-1)) view_name ' (' view_type ') ' FROM uvtemp CONNECT BY PRIOR view_type = superview_type; DROP TABLE uvtemp; |
| На основе какого запроса было определено представление foo_v? | SET LONG 1000 -- or greater SELECT text FROM user_views WHERE view_name = 'FOO_V'; |
| Какие столбцы присутствуют в представлении foo_v? | SELECT column_name, data_type_mod, data_type FROM user_tab_columns WHERE table_name = 'FOO_V'; |

При работе со словарем данных Oracle могут возникнуть недоразумения из-за того, что объектные таблицы не видны в представлении USER_TABLES. Вместо этого список объектных таблиц присутствует в USER_OBJECT_TABLES (а также USER_ALL_TABLES).

Привилегии

С объектными типами связана группа привилегий системного уровня:

- CREATE [ANY] TYPE — создание, изменение и удаление объектных типов и тел типов. ANY означает «в любой схеме».
- CREATE [ANY] VIEW — создание и удаление представлений, включая объектные представления. ANY означает «в любой схеме».
- ALTER ANY TYPE — использование средств ALTER TYPE с типами в любой схеме.
- EXECUTE ANY TYPE — использование объектного типа из любой схемы для таких целей, как создание экземпляров, выполнение методов, обращение по ссылкам и размыменование.
- UNDER ANY TYPE — создание подтипа в одной схеме как производного от типа в любой другой схеме.
- UNDER ANY VIEW — создание подпредставления в одной схеме как производного от представления в любой другой схеме.

Всего существуют три разновидности привилегий объектного уровня для объектных типов: EXECUTE, UNDER и DEBUG. Также важно понимать, как традиционные привилегии DML применяются к объектным таблицам и представлениям.

Привилегия EXECUTE

Если вы хотите, чтобы ваш коллега Джо использовал один из ваших типов в своих программах PL/SQL или таблицах, предоставьте ему привилегию EXECUTE:

```
GRANT EXECUTE on catalog_item_t TO joe;
```

Если Джо обладает привилегией, необходимой для создания синонимов, и работает в Oracle9i Database Release 2 и выше, он сможет создать синоним:

```
CREATE SYNONYM catalog_item_t FOR scott.catalog_item_t;
```

и использовать его следующим образом:

```
CREATE TABLE catalog_items OF catalog_item_t;
```

и/или так:

```
DECLARE  
  an_item catalog_item_t;
```

Джо также может использовать уточненную ссылку на тип `scott.catalog_item_t`.

Если вы ссылаетесь на объектный тип в хранимой программе, а потом предоставляете привилегию EXECUTE для этой программы пользователю или роли, наличие этой привилегии для типа не обязательно, даже если программа определяется с правами вызывающего (см. главу 24). Аналогичным образом, если пользователь имеет привилегию DML для представления, содержащего триггер INSTEAD OF для этой операции DML, пользователю не понадобятся явные привилегии EXECUTE, если триггер ссылается на объектный тип, потому что триггеры выполняются с моделью прав создателя. Тем не менее привилегия EXECUTE необходима для пользователей, выполняющих анонимные блоки, в которых используется объектный тип.

Привилегия UNDER

Привилегия UNDER дает право создания подтипов. Она предоставляется следующим образом:

```
GRANT UNDER ON image_t TO scott;
```

Чтобы схема могла создать подтип, необходимо определить супертип с правами вызывающего (AUTHID CURRENT_USER).

Она также может дать получателю право создания подпредставления:

```
GRANT UNDER ON images_v TO scott;
```

Привилегия DEBUG

Если ваш коллега использует отладчик PL/SQL для анализа кода, использующего созданный вами тип, вы можете предоставить ему привилегию DEBUG:

```
GRANT DEBUG ON image_t TO joe;
```

Предоставление этой привилегии позволит другому разработчику заглянуть «за кулисы», изучить переменные, использованные в типе, и установить точки прерывания внутри методов.

Привилегия DEBUG также применяется к объектным представлениям, давая возможность отладки исходного кода PL/SQL триггеров INSTEAD OF.

Привилегии DML

Для объектных таблиц традиционные привилегии `SELECT`, `INSERT`, `UPDATE` и `DELETE` также имеют смысл. Пользователь, обладающий только привилегией `SELECT` для объектной таблицы, может загрузить любые реляционные столбцы базового типа, на котором основана таблица, но ему не удастся загрузить объект как объект (то есть `VALUE`, `TREAT`, `REF` и `DEREF` ему недоступны). Другие привилегии DML — `INSERT`, `UPDATE` и `DELETE` — также применяются только к реляционной интерпретации таблицы.

Аналогичным образом получатель не сможет использовать конструктор или другие объектные методы, если только владелец типа объекта не предоставит ему привилегию `EXECUTE` для объектного типа. Все столбцы, определенные в подтипах, останутся невидимыми.

О целесообразности применения объектно-ориентированного подхода

За прошедшие годы стало ясно, что ни один стиль программирования не имеет монополии на такие фундаментальные и важные моменты, как соответствие требованиям, эффективность и производительность, удобство разработки и надежность системы. Нам довелось видеть множество восторгов, преувеличенных ожиданий, модных тенденций, горячего энтузиазма — всего этого в свое время было вдоволь и в отношении объектно-ориентированных технологий. Нельзя сказать, что данная технология не помогает решать определенные проблемы, но это не панацея, как думают многие.

Возьмем, к примеру, принцип объектной декомпозиции, используемый при разработке объектных иерархий. При тщательном моделировании объектов реального мира получают удобные и понятные программные компоненты, из которых легко собираются даже крупномасштабные системы. Звучит многообещающе, не правда ли? Однако существует множество способов декомпозиции объектов реального мира, и эти объекты редко удается представить в виде простой иерархии. Например, декомпозицию нашего библиотечного каталога можно выполнить на основе, скажем, носителей (печатные материалы, аудиозаписи, цифровой формат и т. д.). Хотя Oracle предоставляет прекрасные возможности для эволюции типов, серьезные изменения в иерархии типов ведут к таким тяжелым последствиям, что на них обычно никто не решается. И не программные средства тому виной.

Нельзя сказать, что объединение программной логики (методов) и данных (атрибутов) дает какие-либо очевидные преимущества. Пока никто еще убедительно не доказал, что это лучше, чем хранение структур данных (логической и физической структуры таблиц) отдельно от процессов (процедур, функций и пакетов). И многие признают, что структуры данных реальных компаний меняются гораздо реже, чем алгоритмы работы с этими данными. Даже среди проектировщиков ООП-систем общепризнанным является тот факт, что изменяющиеся элементы системы следует отделять от более стабильных.

Последнее обстоятельство особенно интересно. Ярые приверженцы объектного подхода, которые настаивают на объединении данных и операций, в то же время подчеркивают важность подхода «модель—представление—контроллер», «отделяющего бизнес-логику от данных». Вам не кажется, что в этом есть некое противоречие?

Многие сторонники ООП главным преимуществом данного подхода считают возможность многократного использования кода. Это говорилось столько раз, что должно было уже стать правдой! К сожалению, мало кто потрудился привести убедительные доказательства, так как не существует четко определенного понятия «многократного

использования». Даже приверженцы объектного подхода признают многократное использование возможным в первую очередь для компонентов высших уровней, поскольку объекты обычно предназначаются для узкоспециализированного применения. Наш опыт показывает, что возможность повторного использования объектного кода ничуть не выше, чем у обычных хорошо спроектированных программ.

Безусловно, объектно-ориентированные методы можно применять в PL/SQL для повторного использования программного кода. Наш соавтор Дон Бейлз (Don Bales), известный специалист по объектно-ориентированному программированию, использовал пакеты PL/SQL как «типы» около 10 лет, и он говорит, что ему удавалось брать целые пакеты (и сопровождающие таблицы) и переносить их в новые проекты без каких-либо изменений. Он считает, что «недостающим звеном» большинства объектных методов является точная модель человека, работающего с программным продуктом, — то есть пользователя. Дон моделирует его как объект с поведением, реализованным в выполняемой программе.

Независимо от выбора методологии разработки, среди ключевых факторов успеха программного продукта следует выделить опыт решения аналогичных задач, возможность привлечения опытных руководителей проектов, а также включение осмысленной фазы проектирования. Использование объектных или любых других методологий скорее приведет к положительному результату, чем неспланированная, хаотично разрастающаяся система.

В завершение приведем несколько рекомендаций, касающихся применения объектных функций Oracle:

- Если вы используете OCI (Oracle Call Interface), то возможность кэширования на стороне клиента и сложной выборки объектов могут склонить вас к интенсивному использованию объектных средств Oracle. Автор не работает с OCI и не может поделиться опытом в этой области.
- Если в вашей организации уже используется объектно-ориентированное программирование, объектные средства Oracle могут облегчить интеграцию технологий баз данных в имеющиеся системы.
- Даже если вы не намерены пользоваться объектным подходом, не отвергайте вместе с ним и коллекции. Помните, что они обладают широчайшими возможностями, а для их применения не нужны ни объектные типы, ни объектные представления.
- Тем, кто никогда не имел дела с объектно-ориентированным программированием, новые объектные возможности Oracle могут показаться сложными, однако время, потраченное на их освоение, не пройдет даром. В частности, попробуйте использовать объектные представления для существующих систем.
- Не стоит отвергать объектные типы и представления из-за неясных впечатлений об их низкой производительности. Oracle постоянно работает над ее повышением. Кроме того, тестирование может показать, что ООП вполне удовлетворяет требованиям приложения.
- Объектно-ориентированные типы используются и в некоторых встроенных средствах Oracle, напрямую не связанных с объектно-ориентированным программированием (прежде всего XML_TYPE, а также Advanced Queuing, Oracle Spatial и Rules Manager). Как неоднократно случалось в прошлом, после того как Oracle начинает пользоваться своей собственной функциональностью, ошибки начинают исправляться быстрее, производительность растет, а рабочие средства становятся более удобными. Так случилось и с объектными типами. Впрочем, важнее другое — для использования этих возможностей нужно иметь хотя бы общее представление об объектных типах, даже если вы сами создавать их не собираетесь.

27

Вызов Java-программ из PL/SQL

Язык Java изначально был спроектирован и продвигался компанией Sun Microsystems. В наши дни его продвигают практически все, кроме Microsoft. Java предоставляет исключительно разнообразный набор программных средств, многие из которых недоступны в PL/SQL напрямую. В этой главе представлены основы создания и использования хранимых процедур Java в Oracle. Также вы узнаете, как создавать и пользоваться функциональностью JSP из PL/SQL.

Oracle и Java

Начиная с Oracle8i Database в поставку Oracle Database Server входит виртуальная машина Java, обеспечивающая эффективное выполнение программ Java в пространстве памяти сервера. Также в комплект поставки Oracle включаются многие базовые библиотеки классов Java, которые становятся не только серьезным оружием в арсенале программиста, но и серьезной темой для книги по PL/SQL. Вот почему в этой книге мы ограничимся следующими целями:

- Предоставить читателю информацию, необходимую для загрузки классов Java в базу данных Oracle, управления этими новыми объектами базы данных и их публикации для использования в PL/SQL.
- Предоставить вводный курс по построению классов Java, чтобы вы смогли создавать простые классы для использования нижележащей функциональности Java.

Как правило, создание и предоставление доступа к хранимым процедурам Java проходит следующим образом:

1. Разработчик пишет исходный код Java. При этом можно использовать любой удобный текстовый редактор или интегрированную среду разработки (например, Oracle JDeveloper).
2. Откомпилируйте классы Java; при желании упакуйте их в файлы `.jar`. Эта операция также может выполняться интегрированной средой или компилятором командной строки Sun `javac`. (Строго говоря, этот шаг не обязателен, потому что вы можете загрузить исходный код в Oracle и воспользоваться встроенным компилятором Java.)
3. Загрузите классы Java в Oracle программой командной строки `loadjava` или командой `CREATE JAVA`.

4. Опубликуйте методы класса Java; для этого напишите «обертки» PL/SQL для вызова кода Java.
5. Предоставьте необходимые привилегии для обертки PL/SQL.
6. Вызовите программы PL/SQL в одной из доступных сред (рис. 27.1).

JSP (Java Stored Procedures)

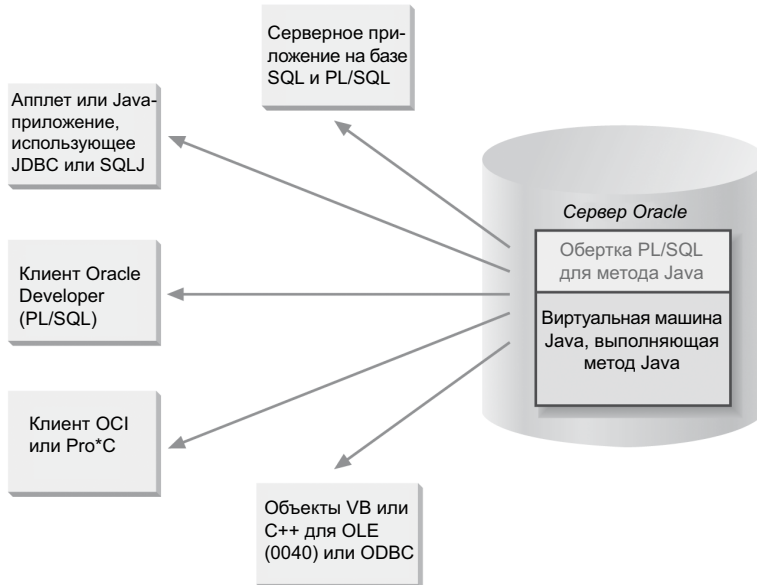


Рис. 27.1. Обращение к JSP из базы данных Oracle

Oracle предоставляет разнообразные компоненты и команды для работы с Java. Некоторые из них перечислены в табл. 27.1.

Таблица 27.1. Компоненты и команды Oracle для Java

| Компонент | Описание |
|---|--|
| Aurora JVM | Виртуальная машина Java, реализованная Oracle в сервере базы данных |
| loadjava | Программа командной строки для загрузки элементов кода Java (классы, файлы .jar и т. д.) в базу данных Oracle |
| dropjava | Программа командной строки для удаления элементов кода Java (классы, файлы .jar и т. д.) из базы данных Oracle |
| CREATE JAVA DROP JAVA, ALTER JAVA | Команды DDL, выполняющие те же операции, что и loadjava и dropjava |
| DBMS_JAVA | Встроенный пакет, содержащий набор инструментов для настройки параметров и других аспектов JVM |

Все эти операции и компоненты будут рассмотрены в этой главе. Более подробную информацию о работе с Java в базе данных Oracle можно найти в книге Дональда Бейлза *Java Programming with Oracle JDBC*. За дополнительной информацией о Java обращайтесь к документации Sun Microsystems и к книгам издательства O'Reilly из серии, посвященной Java (а также к другим книгам, которые я рекомендую далее в этой главе). Более подробная документация об использовании Java с Oracle приведена в учебных руководствах Oracle Corporation.

Подготовка к использованию Java в Oracle

Прежде чем вызывать методы Java из программ PL/SQL, вы должны:

- убедиться в том, что поддержка Java включена в установку Oracle Database Server;
- построить и загрузить элементы кода и классы Java;
- в некоторых случаях — принять меры к тому, чтобы вашей учетной записи пользователя Oracle были предоставлены разрешения, относящиеся к Java.

Установка Java

Поддержка Java может устанавливаться с сервером Oracle — а может и не устанавливаться в зависимости от версии Oracle и решений, принятых администратором базы данных в ходе установки Oracle. Чтобы проверить, установлена ли поддержка Java, выполните следующий запрос:

```
SELECT COUNT(*)  
  FROM all_objects  
 WHERE object_type LIKE 'JAVA%';
```

Если результат равен 0, то поддержка Java определенно не установлена; попросите своего администратора выполнить сценарий с именем `$ORACLE_HOME/javavm/install/initjvm.sql`.

Вы как разработчик, вероятно, захотите строить и тестировать Java-программы на вашей рабочей станции, а для этого вам потребуется пакет JDK (Java Development Kit). У вас есть два варианта: либо загрузите JDK самостоятельно по адресу <http://java.sun.com/>, либо, если вы используете стороннюю среду разработки (такую, как Oracle JDeveloper), возможно, вы сможете использовать прилагаемую версию JDK. Будьте внимательны и обратите особое внимание на точное совпадение номера версии JDK.

При загрузке Java с сайта Sun вам придется выбирать между множеством разных сокращений и версий. Лично у меня остались неплохие впечатления от Java 2 Standard Edition (J2SE) с пакетом Core Java — вместо пакета Desktop, включающего набор ненужных мне средств построения графического интерфейса. Также необходимо выбрать между JDK и JRE (Java Runtime Engine). Всегда выбирайте JDK, если вы собираетесь что-то компилировать! В том, что касается выбора правильной версии, я бы предложил ориентироваться на версию сервера Oracle. В следующей таблице приведены некоторые рекомендации.

| Версия Oracle | Версия JDK |
|------------------------------------|------------|
| Oracle8i Database (8.1.5) | JDK 1.1.6 |
| Oracle8i Database (8.1.6 or later) | JDK 1.2 |
| Oracle9i Database | J2SE 1.3 |
| Oracle Database 10g Release 1 | J2SE 1.4.1 |
| Oracle Database 10g Release 2 | J2SE 1.4.2 |

Если вам приходится поддерживать сразу несколько версий сервера Oracle, выберите последнюю версию и будьте внимательны с используемой функциональностью.

Еще одно неочевидное обстоятельство, о котором необходимо знать: если ваша программа Java не компилируется, убедитесь в том, что в переменную окружения `CLASSPATH` были включены ваши классы, а также классы, предоставляемые Oracle.

Построение и компиляция кода Java

У многих разработчиков PL/SQL (в том числе и у меня) опыт работы с объектно-ориентированными языками оставляет желать лучшего, поэтому освоить Java может быть

непросто. За то непродолжительное время, в течение которого я изучал и использовал Java, я пришел к следующим выводам:

- Освоить синтаксис, необходимый для построения простых классов в Java, не так сложно.
- Начать применять код Java в PL/SQL тоже несложно, но...
- Написание полноценных объектно-ориентированных приложений с использованием Java потребует существенных усилий и изменения менталитета для разработчиков PL/SQL!

О разных аспектах Java написано много (очень, очень много!) книг, среди которых немало просто превосходных. Я рекомендую обратить особое внимание на следующие книги:

- *The Java Programming Language*, авторы Кен Арнольд (Ken Arnold), Джеймс Гослинг (James Gosling) и Дэвид Холмс (David Holmes) (издательство Addison-Wesley).

Джеймс Гослинг — создатель языка Java, поэтому, как и следовало ожидать, книга весьма полезна. Она написана простым, понятным языком и хорошо обучает основам языка.

- *Java in a Nutshell*, автор Дэвид Фленаган (David Flanagan) (издательство O'Reilly). Эта очень популярная и часто обновляемая книга содержит короткий, но превосходный вводный курс. За ним следует краткий справочник всех основных элементов языка, удобный и снабженный множеством перекрестных ссылок.
- *Философия Java*, автор Брюс Экель (издательство «Питер»).

В книге доступно и нестандартно излагаются концепции объектно-ориентированного программирования (кстати, ее можно бесплатно загрузить с сайта MindView). Если вам близок стиль изложения моей книги, то и *Философия Java* наверняка понравится.

Позднее в этой главе, когда я буду демонстрировать вызов методов Java из PL/SQL, также будет приведено пошаговое описание создания относительно простых классов. Во многих ситуациях этого обсуждения будет вполне достаточно для решения практических задач.

Назначение разрешений для разработки и выполнения Java-кода

До выхода версии 8.1.6 в Oracle использовалась несколько иная модель безопасности Java, поэтому ниже две модели будут рассмотрены по отдельности.

Безопасность Java для Oracle (версии, предшествующие 8.1.5)

В ранних выпусках Oracle8i Database (до версии 8.1.6) поддерживалась относительно простая модель безопасности Java. По сути было всего две роли базы данных, которые мог предоставить администратор базы данных:

- `JAVAUSERPRIV` — относительно небольшой набор разрешений Java, включая просмотр свойств.
- `JAVASYSPRIV` — самые важные разрешения, включая обновление защищенных пакетов.

Например, чтобы разрешить Скотту выполнение любых операций, связанных с Java, я должен был выдать следующую команду из учетной записи `SYSDBA`:

```
GRANT JAVAUSERPRIV TO scott;
```

А если я хотел ограничить возможности операций, которые он мог выполнять с Java, выполнялась следующая команда:

```
GRANT JAVAUSERPRIV TO scott;
```

Например, для создания файла средствами Java потребуется роль `JAVASYSPRIV`; для чтения или записи в файл достаточно роли `JAVAUSERPRIV`. За дополнительной информацией о том, какие привилегии Java соответствуют разным ролям Oracle, обращайтесь к руководству *Oracle Java Developer's Guide*.

При инициализации виртуальная машина Java устанавливает экземпляр `java.lang.SecurityManager` (менеджер безопасности Java). Oracle использует их в сочетании со средствами безопасности Oracle Database для определения того, кому разрешен вызов тех или иных методов Java.

Если пользователь, не обладающий достаточными привилегиями, попытается выполнить недопустимую операцию, виртуальная машина Java инициирует исключение `java.lang.SecurityException`. А вот что вы увидите в SQL*Plus:

```
ORA-29532: Java call terminated by uncaught Java exception:
        java.lang.SecurityException
```

Выполнение методов Java в базе данных может создать различные проблемы, связанные с безопасностью, — особенно при взаимодействиях с файловой системой на стороне сервера или другими ресурсами операционной системы. При проверке операций ввода/вывода Oracle следует двум правилам:

- Если динамическому идентификатору была назначена привилегия `JAVASYSPRIV`, то менеджер безопасности разрешает выполнение операции.
- Если динамическому идентификатору была назначена привилегия `JAVAUSERPRIV`, то менеджер безопасности проверяет действительность операции по тем же правилам, что и для пакета PL/SQL `UTL_FILE`. Другими словами, файл должен находиться в каталоге (или подкаталоге), заданном параметром `UTL_FILE_DIR` в файле инициализации базы данных.

Безопасность Java в Oracle (версия 8.1.6 и выше)

Начиная с версии 8.1.6 в виртуальной машине Java в Oracle появилась поддержка безопасности Java 2, в которой разрешения предоставляются на уровне классов — гораздо более сложный и точный механизм управления безопасностью. В этом разделе приводятся примеры, которые показывают, какой код, связанный с безопасностью, вам придется писать (за дополнительными подробностями и примерами обращайтесь к документации Oracle).

Обычно для предоставления разрешений используется процедура `DBMS_JAVA.GRANT_PERMISSION`. В следующем примере эта программа вызывается для предоставления схеме `BATCH` разрешений чтения и записи файла `lastorder.log`:

```
/* Необходимо подключение с правами администратора базы данных */
BEGIN
    DBMS_JAVA.grant_permission(
        grantee => 'BATCH',
        permission_type => 'java.io.FilePermission',
        permission_name => '/apps/OE/lastorder.log',
        permission_action => 'read,write');
END;
/
COMMIT;
```

Помните, что при таких вызовах получатель привилегий должен записываться в верхнем регистре; в противном случае Oracle не найдет имя учетной записи.

Также обратите внимание на `COMMIT`. Оказывается, вызов `DBMS_JAVA` постоянно записывает данные разрешений в таблицу в словаре данных Oracle, но эта операция не закрепляется автоматически. Для получения данных разрешений можно воспользоваться представлениями `USER_JAVA_POLICY` и `DBA_JAVA_POLICY`.

Следующая последовательность команд сначала предоставляет разрешение для обращения к файлам в каталоге, а затем ограничивает разрешения конкретным файлом:

```
BEGIN
/* Сначала разрешения на чтение и запись предоставляются всем */
DBMS_JAVA.grant_permission(
    'PUBLIC',
    'java.io.FilePermission',
    '/shared/*',
    'read,write');

/* Затем встроенная процедура отзывает разрешения
   чтения и записи для конкретного файла у всех
*/
DBMS_JAVA.restrict_permission(
    'PUBLIC',
    'java.io.FilePermission',
    '/shared/secretfile',
    'read,write');

/* Теперь ограничение переопределяется так, чтобы пользователь
   | мог читать и записывать этот файл
*/
DBMS_JAVA.grant_permission(
    'BOB',
    'java.io.FilePermission',
    '/shared/secretfile',
    'read,write');

COMMIT;
END;
```

Набор предопределенных разрешений, которые предоставляет Oracle:

```
java.util.PropertyPermission
java.io.SerializablePermission
java.io.FilePermission
java.net.NetPermission
java.net.SocketPermission
java.lang.RuntimePermission
java.lang.reflect.ReflectPermission
java.security.SecurityPermission
oracle.aurora.rdbms.security.PolicyTablePermission
oracle.aurora.security.JServerPermission
```

Oracle также поддерживает механизмы Java для создания ваших разрешений; за подробностями обращайтесь к руководству *Oracle Java Developer's Guide*.

Простая демонстрация

Прежде чем углубляться в подробности, рассмотрим все действия, необходимые для работы с Java из PL/SQL. Попутно я представлю основные технологические компоненты, необходимые для решения этой задачи.

Допустим, я хочу удалить файл из PL/SQL. До выхода Oracle8i Database это делалось так:

- В Oracle7 (7.3 и ранее) я отправлял сообщение в канал базы данных; далее программа-слушатель на С получала сообщение («Удалить файл X») и выполняла всю работу.
- В Oracle8 и выше я создавал библиотеку, которая ссылалась на DLL-библиотеку языка С или общую библиотеку. Далее из кода PL/SQL вызывалась программа в этой библиотеке, которая и удаляла файл.

Решение с каналами несложно, но это неуклюжий обходной маневр. Решение с внешней процедурой в Oracle8 Database лучше, но его тоже нельзя назвать прямолинейным, особенно если вы не знаете языка С. Похоже, Java-решение лучше всех остальных. Хотя

оно требует некоторых базовых знаний Java и существенно меньшей квалификации, чем для написания эквивалентного кода С. В поставку Java включаются готовые классы, предоставляющие компактные и удобные программные интерфейсы для разнообразных функций, включая файловый ввод/вывод.

Основные действия, выполняемые в этой демонстрации:

1. Определение используемой функциональности Java.
2. Построение собственного класса для того, чтобы иметь возможность вызова функций Java через PL/SQL.
3. Компиляция класса и загрузка его в базу данных.
4. Построение программы PL/SQL для вызова созданного мной метода класса.
5. Удаление файлов из кода PL/SQL.

Поиск функциональности Java

Недавно мой редактор Дебора Рассел (Deborah Russell) любезно прислала мне подборку книг по Java издательства O'Reilly. Я взял здоровенный том *Java Fundamental Classes Reference* (авторы Марк Гранд (Mark Grand) и Джонатан Кнудсен (Jonathan Knudsen)) и поискал в алфавитном указателе категорию «Файл». Мое внимание привлекла ссылка «класс File», и я заглянул на указанную страницу.

Там я нашел информацию о классе с именем `java.io.File` — а именно то, что он «предоставляет набор методов для получения информации о файлах и каталогах». И не только для получения информации — класс также содержит методы (процедуры и функции) для удаления и переименования файлов, создания каталогов и т. д. Именно то, что нужно! Ниже приводится часть программного интерфейса класса `File`:

```
public class java.io.File {  
    public boolean delete();  
    public boolean mkdir ();  
}
```

Другими словами, я вызываю логическую функцию Java для удаления файла. Если удаление файла проходит успешно, функция возвращает `TRUE`; в противном случае возвращается `FALSE`.

Построение класса Java

Зачем строить собственный класс Java на базе класса `File`? Почему бы не вызвать эту функцию в обертке PL/SQL? По двум причинам:

- Метод класса Java обычно выполняется для конкретного объекта — *экземпляра*, который создается на основе класса. В PL/SQL я не могу создать объект Java, а затем снова вызвать метод для этого объекта; иначе говоря, PL/SQL позволяет вызывать только статические методы.
- Хотя и в Java, и в PL/SQL существуют логические типы данных (в Java даже существует примитивный тип и класс), они не соответствуют друг другу. Невозможно передать логическое значение из Java напрямую как логическое значение PL/SQL.

Следовательно, я должен построить собственный класс, который будет:

- создавать объект класса `File`;
- выполнять метод `delete` этого объекта;
- возвращать значение, которое правильно интерпретируется в PL/SQL.

Очень простой класс, использующий метод `File.delete`, выглядит так:

```
/* Файл в Сети: JDelete.java */  
import java.io.File;
```



```
public class JDelete {  
  
    public static int delete (String fileName) {  
        File myFile = new File (fileName);  
        boolean retval = myFile.delete();  
        if (retval) return 1; else return 0;  
    }  
}
```

Рисунок 27.2 поясняет смысл каждого шага в этом коде, но суть происходящего ясна: метод `JDelete.delete` просто создает фиктивный объект `File` для заданного имени файла, чтобы я мог вызвать для этого файла метод `delete`. Объявляя метод статическим, я делаю возможным его использование без создания экземпляра. Статические методы связываются с классом, а не с конкретными экземплярами этого класса.

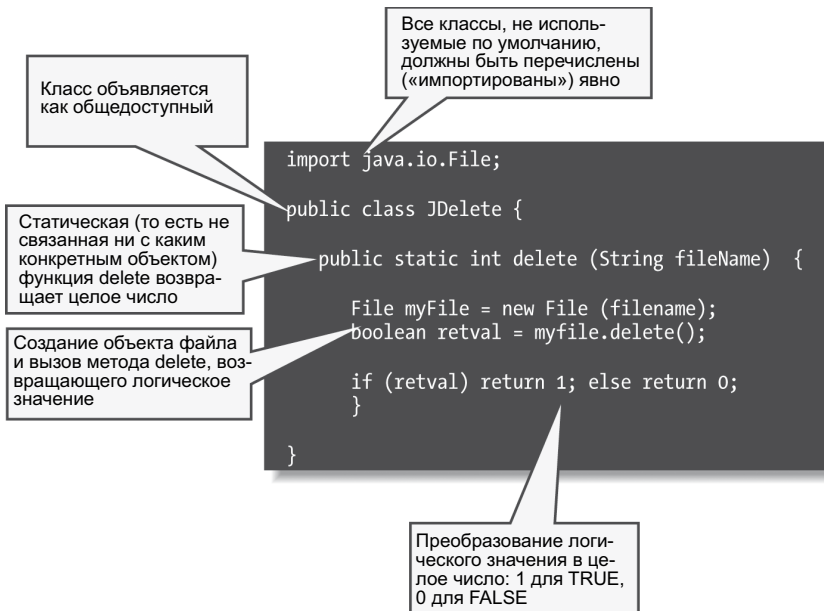


Рис. 27.2. Простой класс Java, предназначенный для удаления файла

Приведенный класс `JDelete` подчеркивает некоторые различия между Java и PL/SQL, о которых следует помнить:

- В Java нет ключевых слов `BEGIN` и `END` для блоков, циклов и условных команд. Вместо этого блок заключается в фигурные скобки.
- В Java учитывается регистр символов: «if» — совсем не то же самое, что «IF».
- В качестве оператора присваивания используется обычный знак равенства (=) вместо составного оператора, используемого в PL/SQL (:=).
- При вызове метода, не получающего аргументов (например, метода `delete` класса `File`), все равно необходимо ставить пару круглых скобок. В противном случае компилятор Java попытается интерпретировать метод как член класса или структуры данных.

В общем-то ничего сложного! Конечно, вы не видели, как я возился с множеством мелких синтаксических ошибок, сталкивался со скрытыми ловушками регистра символов и мучился с настройкой `CLASSPATH`. Представьте сами — и не надейтесь, что в вашей работе всего этого не будет!

Компиляция и загрузка в Oracle

Написанный класс нужно откомпилировать. На компьютере с Microsoft Windows для этого можно было бы открыть консольный сеанс из каталога, в котором хранится исходный код, убедиться в том, что путь к компилятору Java (`javac.exe`) включен в список `PATH`, и выполнить следующую команду:

```
C:\samples\java> javac JDelete.java
```

Если компиляция проходит успешно, компилятор генерирует файл с именем `JDelete.class`. Понятно, что функцию стоит протестировать, прежде чем загружать ее в Oracle и пытаться использовать из PL/SQL. Построение и тестирование всегда желательно проводить в пошаговом режиме. Java предоставляет для этого простое средство: метод `main`. Если вы включите в свой класс метод, не возвращающий значения (то есть процедуру) с именем `main`, и передадите ему правильный список параметров, то вы сможете вызвать свой класс, и при вызове будет выполнен этот код.

Метод `main` — один из примеров особой интерпретации в языке Java некоторых элементов, имеющих правильную сигнатуру. Другим примером служит метод `toString`. Если вы добавите в класс метод с таким именем, он будет автоматически вызываться для вывода вашего пользовательского описания объекта. Данная возможность особенно полезна для классов, состоящих из множества элементов, которые имеют смысл только в определенном представлении или же по иным причинам требуют дополнительного форматирования для восприятия пользователем.

Итак, добавим в `JDelete` простой метод `main`:

```
import java.io.File;

public class JDelete {
    public static int delete (String fileName) {
        File myFile = new File (fileName);
        boolean retval = myFile.delete();
        if (retval) return 1; else return 0;
    }

    public static void main (String args[]) {
        System.out.println (
            delete (args[0])
        );
    }
}
```

Первый элемент массива `args (0)` представляет первый аргумент, переданный из среды вызова. Затем класс следует перекомпилировать:

```
C:\samples\java> javac JDelete.java
```

Если предположить, что каталог с исполняемым файлом `java` включен в список `PATH`, тогда

```
C:\samples\java> java JDelete c:\temp\te_employee.pks
1
```

```
C:\samples\java> java JDelete c:\temp\te_employee.pks
0
```

При первом запуске метода `main` выводится значение 1 (`TRUE`), которое сообщает, что файл был удален. Поэтому вполне логично, что при повторном выполнении той же команды `main` выводит 0. Трудно удалить файл, который уже был удален.

Мой класс компилируется, и я проверил, что метод `delete` работает — теперь можно загрузить его в схему `SCOTT` базы данных Oracle при помощи команды Oracle `loadjava`.



И еще одно преимущество Java перед PL/SQL: если в PL/SQL для вывода строки приходится вводить целых 20 символов (DBMS_OUTPUT.PUT_LINE), в Java достаточно всего 18 (System.out.println). Проектировщики языков, пожалейте нас! Впрочем, Алекс Романкевич, один из наших технических рецензентов, замечает, что если включить в начало класса объявление «private static final PrintStream o = System.out;», то данные можно будет выводить командой o.println — всего девять символов!

Программа `loadjava` включается в поставку Oracle, поэтому путь к ней уже должен быть включен в список `PATH`, если на вашей локальной машине установлен сервер или клиент Oracle:

```
C:\samples\java> loadjava -user scott/tiger -oci8 -resolve JDelete.class
```

Я даже могу убедиться в том, что класс загружен, запросив содержимое словаря данных `USER_OBJECTS` при помощи вспомогательной программы, которая будет представлена позднее в этой главе:

```
SQL> EXEC myjava.showobjects
```

| Object Name | Object Type | Status | Timestamp |
|-------------|-------------|--------|------------------|
| JDelete | JAVA CLASS | VALID | 2005-05-06:15:01 |

Собственно, это все, что относится исключительно к Java. Теперь мы можем вернуться в уютный мир PL/SQL.

Построение обертки PL/SQL

Наша задача — сделать так, чтобы каждый подключившийся к моей базе данных смог удалять файлы из PL/SQL без лишних хлопот. Для этого я создам обертку PL/SQL, которая внешне напоминает функцию PL/SQL, но в действительности попросту передает управление коду Java:

```
/* Файл в Сети: fdelete.sf */
FUNCTION fDelete (
    file IN VARCHAR2)
    RETURN NUMBER
AS LANGUAGE JAVA
    NAME 'JDelete.delete (
        java.lang.String)
        return int';
```

Реализация функции `fdelete` состоит из строки, описывающей вызов метода Java. Список параметров должен соответствовать списку параметров метода, но на месте каждого параметра указывается полностью уточненный тип данных. В данном случае это означает, что вместо простого «String» я должен указать полное имя пакета Java, содержащего класс `String`. В секции `RETURN` просто указывается `int` для целочисленного значения. Поскольку `int` является примитивным типом данных, а не классом, такое имя является полной спецификацией.

Небольшое отступление: я также мог написать спецификацию вызова для процедуры, вызывающей метод `JDelete.main`:

```
PROCEDURE fDelete2 (
    file IN VARCHAR2)
AS LANGUAGE JAVA
    NAME 'JDelete.main(java.lang.String[])';
```

Метод `main` отличается от других методов; хотя он получает массив объектов `String`, спецификация вызова может определяться с любым количеством параметров.

Удаление файлов из PL/SQL

Итак, я компилирую функцию и готовлюсь выполнить почти что волшебный, некогда трудный (а то и невозможный) трюк:

```
SQL> @fdelete.sf
```

Function created.

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
```

И вот что я получаю:

ERROR at line 1:

ORA-29532: Java call terminated by uncaught Java exception: java.security.

```
AccessControlException: the
Permission (java.io.FilePermission c:\temp\te_employee.pkb delete) has not been
granted to BOB. The PL/SQL
to grant this is dbms_java.grant_permission( 'BOB', 'SYS:java.io.FilePermission',
'c:\temp\te_employee.pkb', 'delete' )
ORA-06512: at "BOB.FDELETE", line 1
ORA-06512: at line 1
```

Я забыл предоставить себе необходимые разрешения! Но взгляните на сообщение — Oracle любезно сообщает мне не только суть проблемы, но и возможный способ ее исправления. Я прошу своего знакомого администратора выполнить команду следующего вида (небольшая вариация на тему предложения Oracle):

```
CALL DBMS_JAVA.grant_permission(
'BOB',
'SYS:java.io.FilePermission',
'c:\temp\*',
'read,write,delete' );
```

А теперь результат выглядит так:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
1
SQL> exec DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
0
```

Ура, заработало!

Эта функция может стать основой для построения служебных программ. Как насчет процедуры, удаляющей все файлы, найденные в строках вложенной таблицы? Или и того лучше — как насчет процедуры, которая получает имя каталога и фильтр (например, «все файлы с именами вида *.tmp») и удаляет из каталога все файлы, соответствующие фильтру?

Конечно, на практике мне следовало бы построить пакет и разместить в нем всю замечательную новую функциональность. Но сначала давайте повнимательнее присмотримся к тому, что я уже сделал.

Использование loadjava

Программа `loadjava` работает в режиме командной строки и загружает файлы Java в базу данных. При первом выполнении для схемы `loadjava` создает несколько объектов в локальной схеме. Хотя точный список зависит от версии Oracle, скорее всего, в табличном пространстве по умолчанию вы найдете следующее:

- `CREATE$JAVA$LOB$TABLE` — таблица с элементами кода Java, создаваемая в каждой схеме. Каждый новый класс, загружаемый командой `loadjava`, создает одну строку в этой таблице, с сохранением байтов класса в столбце `BLOB`.

- SYS_C ... (точное имя зависит от версии) — уникальный индекс в упомянутой выше таблице.
 - SYS_IL ... (точное имя зависит от версии) — индекс LOB в упомянутой выше таблице.
- Кстати говоря, если вы не располагаете достаточными привилегиями или квотами для создания этих объектов в табличном пространстве по умолчанию, операция загрузки завершится неудачей.

Прежде чем выполнять загрузку, Oracle сначала проверяет, существует ли загружаемый объект и изменяется ли он. Тем самым сводятся к минимуму лишние повторные загрузки и объявление недействительными зависимых классов¹.

Затем операция загрузки вызывает команду DDL CREATE JAVA для загрузки классов Java из столбца BLOB таблицы CREATE\$JAVA\$LOB\$TABLE в РСУБД в виде объектов схемы. Загрузка происходит только при выполнении следующих условий:

- Класс загружается в первый раз.
- Класс изменился.
- При вызове задан ключ -force.

Базовый синтаксис вызова:

```
loadjava {-user | -u} пользователь/пароль[@база_данных]
[ключ ...] filename [имя_файла]...
```

Здесь *имя_файла* — файл с исходным кодом Java, SQL J, класса, .jar, ресурса, свойств или .zip. Например, следующая команда загружает класс JFile в схему SCOTT:

```
C:> loadjava -user scott/tiger -oci8 -resolve JFile.class
```

Несколько слов о loadjava: для вывода справки используется следующий синтаксис:

```
loadjava {-help | -h}
```

В списке ключей или файлов имена могут разделяться только пробелами:

```
-force, -resolve, -thin // Нет
-force -resolve -thin   // Да
```

Однако в списке пользователей или ролей имена должны разделяться только запятыми:

```
SCOTT, PAYROLL, BLAKE // Нет
SCOTT,PAYROLL,BLAKE  // Да
```

Программа loadjava поддерживает более 40 ключей командной строки; важнейшие ключи перечислены в табл. 27.2.

Таблица 27.2. Часто используемые ключи командной строки loadjava

| Ключ | Описание |
|-----------|---|
| -debug | Предназначается для отладки самого сценария loadjava, а не вашего кода; используется редко |
| -definer | Указывает, что методы загруженных классов будут выполняться с привилегиями создателя, а не вызывающего. (По умолчанию методы выполняются с привилегиями вызывающего.) Разные создатели могут обладать разными привилегиями, а приложение может содержать много классов, поэтому программист должен проследить за тем, чтобы методы заданного класса выполнялись только с минимумом необходимых привилегий |
| -encoding | Задает (или сбрасывает) ключ -encoding в таблице JAVA\$OPTIONS; значение должно быть именем стандартной схемы кодировки JDK (по умолчанию "latin1"). Компилятор использует это значение, так что кодировка загружаемых исходных файлов должна соответствовать заданной. О создании и использовании этого объекта рассказано в разделе «GET_, SET_ и RESET_COMPILER_OPTION: чтение и запись параметров компилятора» (см. с. 973) |

продолжение ⇨

¹ Oracle проверяет контрольную сумму MD5 входного класса и сравнивает ее с контрольной суммой существующего класса.

Таблица 27.2 (продолжение)

| Ключ | Описание |
|-----------|--|
| -force | Обеспечивает принудительную загрузку файлов классов Java независимо от того, изменялись ли они с момента последней загрузки. Учтите, что принудительная загрузка файла класса невозможна, если до этого вы загрузили исходный файл (и наоборот) — сначала необходимо удалить ранее загруженный объект |
| -grant | Предоставляет привилегию EXECUTE для загруженных классов перечисленным пользователям или ролям. (Чтобы вызывать методы класса напрямую, пользователи должны иметь привилегию EXECUTE.) Ключ действует с накоплением: новые пользователи и роли добавляются в список имеющих привилегию EXECUTE. Чтобы отозвать привилегию, либо удалите и перезагрузите объект схемы без -grant, либо воспользуйтесь командой SQL REVOKE. Чтобы предоставить привилегию для объекта схеме другого пользователя, необходимо иметь привилегию CREATE PROCEDURE WITH GRANT |
| -oci8 | Приказывает loadjava взаимодействовать с базой через драйвер OCI JDBC. Этот ключ (используемый по умолчанию) и -thin являются взаимоисключающими. При вызове loadjava с клиентского компьютера без установки Oracle используйте ключ -thin |
| -resolve | После того как все файлы в командной строке будут загружены и откомпилированы (в случае необходимости), разрешает все внешние ссылки в этих классах. Если ключ не указан, файлы загружаются, но не компилируются и не обрабатываются до времени выполнения. Ключ используется для компиляции и разрешения класса, который был загружен ранее. Указывать ключ -force не обязательно, потому что разрешение происходит независимо после загрузки |
| -resolver | Связывает созданные объекты схемы класса со спецификацией определителя, заданной пользователем. Так как спецификация содержит пробелы, она должна быть заключена в двойные кавычки. Этот ключ и -oracleresolver (используется по умолчанию) являются взаимоисключающими |
| -schema | Присваивает созданные объекты схемы Java заданной схеме. Если ключ не задан, используется схема logon. Для загрузки в схему другого пользователя необходимо иметь привилегию CREATE ANY PROCEDURE |
| -synonym | Создает общедоступный синоним для загруженных классов, чтобы они были доступны за пределами схемы, в которой они загружаются. Для задания этого ключа необходимо иметь привилегию CREATE PUBLIC SYNONYM. Если задать этот ключ для исходных файлов, он также применяется к классам, полученным в результате компиляции этих исходных файлов |
| -thin | Приказывает loadjava взаимодействовать с базой данных через «тонкий» (минимальный) драйвер JDBC. Этот ключ и -oci8 (используется по умолчанию) являются взаимоисключающими. При вызове loadjava с клиентского компьютера без установки Oracle используйте ключ -thin |
| -verbose | Включает режим расширенного вывода, в котором выводятся сообщения о ходе выполнения. Очень полезно! |

Как вы, вероятно, понимаете, при работе с `loadjava` приходится учитывать много разных нюансов — например, загрузку отдельных классов или сжатых групп элементов в файлах `.zip` или `.jar`. В документации Oracle приводится более подробная информация о программе `loadjava`.

Программа dropjava

Программа `dropjava` по своему назначению противоположна `loadjava`: она преобразует имена файлов в имена объектов схемы, а затем удаляет объекты схемы вместе со всеми ассоциированными данными. Удаление класса приводит к тому, что классы, зависящие от него прямо или косвенно, становятся недействительными. Удаление объекта исходного кода также приводит к удалению классов, созданных на его основе.

Синтаксис практически идентичен синтаксису `loadjava`:

```
dropjava {-user | -u} пользователь/пароль[@база_данных]
        [ключ ...] filename [имя_файла]...
```

Здесь допустимыми ключами являются `-oci8`, `-encoding` и `-verbose`.

Управление Java в базе данных

В этом разделе более подробно рассматриваются вопросы, связанные с хранением элементов Java в базе данных и управлением этими элементами.

Пространство имен Java в Oracle

Oracle хранит каждый класс Java в базе данных как объект схемы. Имя объекта строится на основе полностью уточненного имени класса (хотя и не совпадает с ним) и включает имена всех вмещающих пакетов. Например, полное имя класса `OracleSimpleChecker` выглядит так:

```
oracle.sqlj.checker.OracleSimpleChecker
```

В базе данных полное имя объекта схемы Java будет выглядеть так:

```
oracle/sqlj/checker/OracleSimpleChecker
```

Иначе говоря, при сохранении в базе данных Oracle точки замещаются косой чертой.

Имя объекта в Oracle, будь то имя таблицы базы данных или класса Java, не может быть длиннее 30 символов. В Java такого ограничения нет; в этом языке допускаются гораздо более длинные имена. Oracle позволяет загрузить в базе данных Oracle класс Java с именем, содержащим до 4000 символов. Если имя элемента Java содержит более 30 символов, база данных автоматически генерирует для этого элемента допустимый псевдоним (длиной менее 31 символа).

Вам никогда не придется использовать этот псевдоним в хранимых процедурах. Вместо этого вы можете продолжать использовать «настоящее» имя элемента Java в своем коде. Oracle при необходимости автоматически связывает длинное имя с псевдонимом.

Получение информации о загруженных элементах Java

Когда вы загрузите в базу данных элементы исходного кода, класса и элемента, информация об этих элементах становится доступной в нескольких представлениях словаря данных (табл. 27.3).

Таблица 27.3. Информация о классах в представлениях словаря данных

| Представление | Описание |
|--|--|
| USER_OBJECTS, ALL_OBJECTS, DBA_OBJECTS | Информация об объектах типов JAVA SOURCE, JAVA CLASS и JAVA RESOURCE |
| USER_ERRORS, ALL_ERRORS, DBA_ERRORS | Ошибки компиляции, обнаруженные при обработке объектов |
| USER_SOURCE | Исходный код Java, если вы использовали команду CREATE JAVA SOURCE для создания объекта схемы Java |

Следующий запрос выводит информацию обо всех объектах, связанных с Java, в моей схеме:

```
/* Файлы в Сети: showjava.sql, myJava.pkg */  
COLUMN object_name FORMAT A30  
SELECT object_name, object_type, status, timestamp  
FROM user_objects  
WHERE (object_name NOT LIKE 'SYS_%'  
      AND object_name NOT LIKE 'CREATE$%'  
      AND object_name NOT LIKE 'JAVA$%'  
      AND object_name NOT LIKE 'LOADLOB%')  
      AND object_type LIKE 'JAVA %'  
ORDER BY object_type, object_name;
```

Секция WHERE отфильтровывает объекты, созданные Oracle для управления объектами Java. Хранимая информация может использоваться разными способами. Ниже приведен пример вывода пакета myjava, доступного на сайте книги:

```
SQL> EXEC myJava.showObjects
Object Name          Object Type   Status   Timestamp
-----
DeleteFile           JAVA CLASS   INVALID  0000-00-00:00:00
JDelete              JAVA CLASS   VALID    2005-05-06:10:13
book                 JAVA CLASS   VALID    2005-05-06:10:07
DeleteFile           JAVA SOURCE   INVALID  2005-05-06:10:06
book                 JAVA SOURCE   VALID    2005-05-06:10:07
```

Следующая команда выводит список всех элементов Java, имена которых содержат строку «Delete»:

```
SQL> EXEC myJava.showobjects ('%Delete%')
```

Столбец USER_OBJECTS.object_name содержит полные имена объектов Java в схеме, если только эти имена не содержат более 30 символов или непреобразуемые символы из набора Юникод. В обоих случаях в столбце object_name выводится короткое имя. Для преобразования коротких имен в длинные используется функция LONGNAME из пакета DBMS_JAVA, который рассматривается в следующем разделе.

Пакет DBMS_JAVA

Встроенный пакет Oracle DBMS_JAVA предоставляет средства для изменения различных характеристик виртуальной машины Java в базе данных.

Пакет DBMS_JAVA содержит множество программ, многие из которых предназначены для внутреннего использования Oracle. Однако мы можем пользоваться другими полезными программами, вызывая их в командах SQL. В табл. 27.4 приведена сводка некоторых программ DBMS_JAVA. Как упоминалось ранее в этой главе, в пакет DBMS_JAVA также включены программы для управления безопасностью и разрешениями.

Таблица 27.4. Часто используемые программы DBMS_JAVA

| Программа | Описание |
|---------------------------------|---|
| Функция LONGNAME | Получает полное (длинное) имя Java для заданного короткого имени Oracle |
| Функция GET_COMPILER_OPTION | Ищет значение параметра в таблице параметров Java |
| Процедура SET_COMPILER_OPTION | Записывает значение параметра в таблицу параметров Oracle и создает таблицу, если она не существует |
| Процедура RESET_COMPILER_OPTION | Сбрасывает значение параметра в таблице параметров Oracle |
| Процедура SET_OUTPUT | Перенаправляет вывод Java в текстовый буфер DBMS_OUTPUT |
| Процедура EXPORT_SOURCE | Экспортирует объект исходного кода Java в Oracle LOB |
| Процедура EXPORT_RESOURCE | Экспортирует объект ресурса Java в Oracle LOB |
| Процедура EXPORT_CLASS | Экспортирует объект класса Java в Oracle LOB |

Эти программы более подробно рассматриваются ниже.

LONGNAME: преобразование длинных имен Java

Длина имен классов Java легко может превысить максимальную длину идентификатора SQL, равную 30 символам. В таких случаях Oracle создает для элемента кода Java уникальное «короткое имя» и использует его для обращений, связанных с SQL и PL/SQL.

Используйте следующую функцию для получения полного (длинного) имени, соответствующего заданному короткому имени:

```
FUNCTION DBMS_JAVA.LONGNAME (shortname VARCHAR2) RETURN VARCHAR2
```

Следующий запрос выводит длинные имена всех классов Java, определенных в текущей схеме, у которых длинные имена не совпадают с короткими:

```
/* Файл в Сети: longname.sql */
SELECT object_name shortname,
       DBMS_JAVA.LONGNAME (object_name) longname
FROM USER_OBJECTS
WHERE object_type = 'JAVA CLASS'
AND object_name != DBMS_JAVA.LONGNAME (object_name);
```

Этот запрос также включен в пакет myJava (файл myJava.pkg). Предположим, я определяю класс со следующим именем:

```
public class DropAnyObjectIdentifiedByTypeAndName {
```

Для Oracle это имя получается слишком длинным. Я могу убедиться в том, что Oracle создает свое короткое имя:

```
SQL> EXEC myJava.showLongnames
```

```
Short Name | Long Name
```

```
-----
Short: /247421b0_DropAnyObjectIdentif
Long: DropAnyObjectIdentifiedByTypeAndName
```

GET_, SET_ и RESET_COMPILER_OPTION: чтение и запись параметров компилятора

Вы можете задать значения некоторых параметров компилятора в таблице базы данных JAVA\$OPTIONS (далее именуемой таблицей параметров). Хотя поддерживается более 40 параметров командной строки, только три из них могут сохраняться в таблице параметров:

- **encoding** — кодировка, в которой записан исходный код. Если значение не задано, компилятор использует значение по умолчанию, которое возвращается методом `Java System.getProperty("file.encoding")`, — например, `ISO646-US`.
- **online** — `true` или `false`; значение применимо только к исходному коду SQLJ. Значение по умолчанию `true` обеспечивает проверку online-семантики.
- **debug** — `true` или `false`; значение `true` эквивалентно использованию `javac -g`. Если параметр не задан, компилятор по умолчанию использует значение `true`.

Если значения параметров не заданы в командной строке `loadjava`, компилятор ищет их в таблице параметров.

Значения этих трех параметров могут задаваться следующими функциями и процедурами DBMS_JAVA:

```
FUNCTION DBMS_JAVA.GET_COMPILER_OPTION (
    объект VARCHAR2, имя_параметра VARCHAR2)
```

```
PROCEDURE DBMS_JAVA.SET_COMPILER_OPTION (
    объект VARCHAR2, имя_параметра VARCHAR2, значение VARCHAR2)
```

```
PROCEDURE DBMS_JAVA.RESET_COMPILER_OPTION (
    объект VARCHAR2, имя_параметра VARCHAR2)
```

Здесь *объект* — имя пакета Java, полное имя класса или пустая строка. После поиска в таблице параметров компилятор выбирает строку, в которой значение *объект* ближе всего соответствует полному имени объекта схемы. Если *объект* — пустая строка, она совпадает с именем любого объекта схемы. Параметр *имя_параметра* определяет имя

задаваемого параметра. Изначально в схеме не существует таблицы параметров. Чтобы создать ее, используйте процедуру `DBMS_JAVA.SET_COMPILER_OPTION` для задания значения. Если таблица не существует, процедура создает ее. Параметры заключаются в апострофы, как показано в следующем примере:

```
SQL> DBMS_JAVA.SET_COMPILER_OPTION ('X.sqlj', 'online', 'false');
```

SET_OUTPUT: включение вывода из Java

При выполнении из базы данных Oracle классы `System.out` и `System.err` передают свой вывод в текущие файлы трассировки, обычно находящиеся в серверном подкаталоге `udump`. Это не самое удобное место, если вы просто хотите протестировать свой код и убедиться в правильности его работы. Пакет `DBMS_JAVA` предоставляет процедуру, вызов которой перенаправляет вывод в текстовый буфер `DBMS_OUTPUT` для автоматического вывода на экран `SQL*Plus`. Процедура имеет следующий синтаксис:

```
PROCEDURE DBMS_JAVA.SET_OUTPUT (buffersize NUMBER);
```

Пример использования этой программы:

```
/* Файл в Сети: ssoo.sql */
SET SERVEROUTPUT ON SIZE 1000000
CALL DBMS_JAVA.SET_OUTPUT (1000000);
```

Передача любого целого числа `DBMS_JAVA.set_output` приводит к его включению. Документации по взаимодействию между этими двумя командами явно недостаточно, но мое тестирование продемонстрировало следующее поведение:

- Минимальный размер буфера (используемый по умолчанию) составляет всего 2000 байт; максимальный размер равен 1 000 000 байт, по крайней мере в Oracle Database 10g Release 1. Передача значения за пределами указанного диапазона не вызывает ошибки; максимум автоматически устанавливается равным 1 000 000 (кроме очень больших чисел).
- Размер буфера, заданный `SET SERVEROUTPUT`, заменяет размер, заданный `DBMS_JAVA.SET_OUTPUT`. Другими словами, если указать меньшее значение при вызове `DBMS_JAVA`, оно будет проигнорировано, и в программе будет использован больший размер.
- Если вы используете Oracle Database 10g Release 2 и выше и размер `SERVEROUTPUT` задан неограниченным (`UNLIMITED`), то максимальный размер буфера Java тоже неограничен.
- Если вывод в Java превышает размер буфера, вы получите ошибку, которую выдает `DBMS_OUTPUT`, а именно:

```
ORA-10027: buffer overflow, limit of nnn bytes
```

Вместо этого вывод усекается по заданному размеру буфера, а выполнение кода продолжается.

Как и в случае с `DBMS_OUTPUT`, вы не увидите вывода вызовов Java до завершения хранимой процедуры, в которой они выполняются.

EXPORT_SOURCE, EXPORT_RESOURCE и EXPORT_CLASS: экспортирование объектов схемы

Пакет Oracle `DBMS_JAVA` предоставляет набор процедур для экспортирования исходного кода, ресурсов и классов:

```
PROCEDURE DBMS_JAVA.EXPORT_SOURCE (
    имя VARCHAR2 IN,
    [ blob BLOB IN | clob CLOB IN ]
);
PROCEDURE DBMS_JAVA.EXPORT_SOURCE (
```

```

имя VARCHAR2 IN,
схема VARCHAR2 IN,
[ blob BLOB IN | clob CLOB IN ]
);
PROCEDURE DBMS_JAVA.EXPORT_RESOURCE (
имя VARCHAR2 IN,
[ blob BLOB IN | clob CLOB IN ]
);
PROCEDURE DBMS_JAVA.EXPORT_RESOURCE (
имя VARCHAR2 IN,
схема VARCHAR2 IN,
[ blob BLOB IN | clob CLOB IN ]
);
PROCEDURE DBMS_JAVA.EXPORT_CLASS (
имя VARCHAR2 IN,
blob BLOB IN
);
PROCEDURE DBMS_JAVA.EXPORT_CLASS (
имя VARCHAR2 IN,
схема VARCHAR2 IN,
blob BLOB IN
);

```

Во всех случаях *name* определяет имя экспортируемого объекта схемы Java, *схема* — имя схемы, которой принадлежит объект (если схема не задана, используется текущая схема), а *blob* | *clob* — большой объект, получающий заданный объект схемы Java.

Классы не могут экспортироваться в CLOB, только в BLOB. Кроме того, во внутреннем представлении исходного кода используется формат UTF-8, поэтому этот формат также используется для хранения исходного кода в BLOB.

Следующий прототип процедуры дает представление о том, как экспортирование используется для получения исходного кода объектов схемы Java:

```

/* Файл в Сети: showjava.sp */
PROCEDURE show_java_source (
NAME IN VARCHAR2, SCHEMA IN VARCHAR2 := NULL
)
-- Общие сведения: вывод исходного кода Java (прототип). Автор: Вадим Лоевский.
IS
b          CLOB;
v          VARCHAR2 (2000);
i          INTEGER;
object_not_available EXCEPTION;
PRAGMA EXCEPTION_INIT(object_not_available, -29532);

BEGIN
/* Перемещение исходного кода Java в CLOB. */
DBMS_LOB.createtemporary(b, FALSE );

DBMS_JAVA.export_source(name, NVL(SCHEMA, USER), b);

/* Чтение CLOB в переменную VARCHAR2 и ее вывод. */
i := 1000;
DBMS_lob.read(b, i, 1, v);
DBMS_OUTPUT.put_line(v);
EXCEPTION
/* Если объект с указанным именем не существует, происходит исключение. */
WHEN object_not_available
THEN
IF DBMS_UTILITY.FORMAT_ERROR_STACK LIKE '%no such%object'
THEN
DBMS_OUTPUT.put_line ('Java object cannot be found.' );
END IF;
END;

```

Допустим, после этого я создаю объект исходного кода Java командой CREATE JAVA:

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
    public class Hello {
        public static String hello() {
            return "Hello Oracle World";
        }
    };
```

Исходный код просматривается следующей командой (предполагается, что вывод DBMS_OUTPUT включен):

```
SQL> EXEC show_java_source ('Hello')
public class Hello {
    public static String hello() {
        return "Hello Oracle World";
    }
};
```

Публикация и использование кода Java в PL/SQL

После того как вы напишете свои классы Java и загрузите их в базу данных Oracle, их методы можно будет вызывать из PL/SQL (и SQL) — но только после того, как методы будут «опубликованы» при помощи обертки PL/SQL.

Спецификация вызова

Обертки PL/SQL необходимо построить только для тех методов Java, которые должны быть доступны через интерфейс PL/SQL. Методы Java могут вызывать другие методы Java в виртуальной машине Java напрямую, без оберток. Чтобы опубликовать метод Java, вы пишете *спецификацию вызова* — заголовок программы PL/SQL (функции или процедуры), тело которой в действительности представляет собой вызов метода Java с использованием секции LANGUAGE JAVA. Эта секция включает следующую информацию о методе Java: полное имя, типы параметров и возвращаемый тип. Спецификации вызовов могут определяться как отдельные функции или процедуры, как программы в пакетах или методы объектного типа, с использованием синтаксиса AS LANGUAGE JAVA после заголовка программной единицы:

```
{IS | AS} LANGUAGE JAVA
NAME 'полное_имя_метода (mun_java[, mun_java]...)'
    [return mun_java]';
```

Здесь *mun_java* — либо полное имя типа Java (например, `java.lang.String`), либо примитивный тип (например, `int`). Обратите внимание: спецификация не должна включать имена параметров, только типы.

Секция NAME однозначно идентифицирует инкапсулируемый метод Java. Полное имя Java и параметры спецификации вызова, определяемые по позиции, должны соответствовать параметрам программы. Если метод Java вызывается без аргументов, укажите пустой список параметров.

Несколько примеров:

- Отдельная функция, вызывающая метод (см. выше):

```
FUNCTION fDelete (
    file IN VARCHAR2)
    RETURN NUMBER
AS LANGUAGE JAVA
```

```
NAME 'JDelete.delete (
    java.lang.String)
    return int';
```

- Пакетная процедура, передающая объектный тип в параметре:

```
PACKAGE nat_health_care
IS
    PROCEDURE consolidate_insurer (ins Insurer)
        AS LANGUAGE JAVA
        NAME 'NHC_consolidation.process(oracle.sql.STRUCT)';
END nat_health_care;
```

- Метод объектного типа:

```
TYPE pet_t AS OBJECT (
    name VARCHAR2(100),
    MEMBER FUNCTION date_of_birth (
        name_in IN VARCHAR2) RETURN DATE
    AS LANGUAGE JAVA
    NAME 'petInfo.dob (java.lang.String)
        return java.sql.Timestamp'
);
```

- Отдельная процедура с параметром OUT:

```
PROCEDURE read_out_file (
    file_name IN VARCHAR2,
    file_line OUT VARCHAR2
)
AS
    LANGUAGE JAVA
    NAME 'utils.ReadFile.read(java.lang.String
        ,java.lang.String[])';
```

Правила спецификаций вызовов

Обратите внимание на следующие обстоятельства:

- Спецификация вызова PL/SQL и публикуемый ею метод Java должны находиться в одной схеме.
- Спецификация вызова предоставляет Oracle точку входа верхнего уровня. Соответственно публиковать можно только открытые статические методы, если только вы не определяете метод объектного типа SQL. В этом случае методы экземпляра могут публиковаться как методы-члены этого типа.
- В списке параметров программы PL/SQL, служащей оберткой для вызова метода Java, не могут задаваться значения по умолчанию.
- Методы в объектно-ориентированных языках не могут присваивать значения объектам, передаваемым в аргументах; методы предназначены для применения к объекту, с которым они связаны. Когда вы вызываете метод из SQL или PL/SQL и изменяете значение аргумента, он должен быть объявлен как параметр OUT или IN OUT в спецификации вызова. Соответствующий параметр Java должен быть одноэлементным массивом соответствующего типа.



Значение элемента можно заменить другим объектом Java соответствующего типа, или (только для параметров IN OUT) изменить значение, если это допускает тип Java. В любом случае новое значение передается на сторону вызова. Например, параметр OUT типа NUMBER в спецификации вызова можно связать с параметром Java, объявленным как float[] p, а затем присвоить новое значение p[0].

Функция, объявляющая параметры OUT или IN OUT, не может вызываться из DML-команд SQL.

Отображение типов данных

Ранее в этой главе был приведен очень простой пример обертки PL/SQL — функция удаления файлов, которая передавала значение `VARCHAR2` для параметра `java.lang.String`. Метод Java возвращал значение `int`, которое затем возвращалось секцией `RETURN NUMBER` функции PL/SQL. Это тривиальные примеры *отображения типов данных*, то есть установления соответствия между типом данных PL/SQL и типом данных Java.

При построении спецификации вызова PL/SQL параметры PL/SQL и Java, а также результат функции сопоставляются по позиции и должны иметь совместимые типы данных. В табл. 27.5 перечислены все отображения типов данных, допустимые в настоящее время между PL/SQL и Java. Если вы используете поддерживаемое отображение типов данных, Oracle выполнит преобразование автоматически.

Таблица 27.5. Допустимые отображения типов данных

| Тип SQL | Класс Java | Тип SQL | Класс Java |
|-----------|---|---|--|
| CHAR | oracle.sql.CHAR | BFILE | oracle.sql.BFILE |
| NCHAR | java.lang.String | BLOB | oracle.sql.BLOB oracle.jdbc2.Blob |
| LONG | java.sql.Date | CLOB | oracle.sql.CLOB |
| VARCHAR2 | java.sql.Time | NCLOB | oracle.jdbc2.Clob |
| NVARCHAR2 | java.sql.Timestamp java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal byte, short, int, long, float, double | Пользовательский объектный тип | oracle.sql.STRUCT java.sql.Struct java.sql.SqlData oracle.sql.ORAData |
| DATE | oracle.sql.DATE java.sql.Date java.sql.Time java.sql.Timestamp java.lang.String | Пользовательский тип REF | oracle.sql.REF java.sql.Ref oracle.sql.ORAData |
| NUMBER | oracle.sql.NUMBER java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal byte, short, int, long, float, double | Непрозрачный тип (такой, как XMLType) | oracle.sql.OPAQUE |
| RAW | oracle.sql.RAW | TABLE VARRAY | oracle.sql.ARRAY |
| LONG RAW | byte[] | Пользовательский тип таблицы или VARRAY | java.sql.Array oracle.sql.ORAData |
| ROWID | oracle.sql.CHAR oracle.sql.ROWID java.lang.String | Любой из перечисленных типов SQL types | oracle.sql.CustomDatum oracle.sql.Datum |

Как видите, Oracle поддерживает автоматическое преобразование только для типов данных SQL. Такие типы данных PL/SQL, как `BINARY_INTEGER`, `PLS_INTEGER`, `BOOLEAN` и типы ассоциативных массивов, не поддерживаются. В таких случаях

действия для передачи данных между двумя средами выполнения приходится выполнять вручную. Нестандартные отображения продемонстрированы далее в разделе «Другие примеры»; за более подробными примерами с использованием JDBC обращайтесь к документации Oracle.

Вызов метода Java в SQL

В DML-командах SQL вы можете вызывать написанные вами функции PL/SQL. Также из кода SQL могут вызываться методы Java, «упакованные» в PL/SQL. Однако такие вызовы должны подчиняться определенным правилам:

- Если метод вызывается из команды `SELECT` или из параллелизированной команды `INSERT`, `UPDATE`, `MERGE` или `DELETE`, то он не должен изменять никакие таблицы базы данных.
- Если метод вызывается из команды `INSERT`, `UPDATE`, `MERGE` или `DELETE`, то он не может обращаться с запросами или изменять никакие таблицы базы данных, измененные этой командой.
- Если метод вызывается из команды `SELECT`, `INSERT`, `UPDATE`, `MERGE` или `DELETE`, то он не может выполнять команды управления транзакциями SQL (например, `COMMIT`), команды управления сеансом (например, `SET ROLE`) или команды управления системой (например, `ALTER SYSTEM`). Метод также не может выполнять команды DDL, потому что они автоматически закрепляют операции в сеансе. Учтите, что эти ограничения снимаются, если метод выполняется из блока анонимной транзакции PL/SQL.

Эти ограничения нужны для того, чтобы избежать побочных эффектов, способных нарушить работу команд SQL. Если вы пытаетесь выполнить команду SQL, которая вызывает метод, нарушающий какие-либо из этих правил, при разборе команды SQL произойдет ошибка времени выполнения.

Обработка исключений в Java

С другой стороны, архитектура обработки исключений Java очень близка к PL/SQL, хотя механизм обработки исключений намного мощнее. Java предоставляет базовый класс исключения с именем `Exception`. Все исключения представляют собой объекты, созданные на базе этого класса — или классов, производных от него (расширяющих этот класс). Вы можете передавать исключения в параметрах и работать с ними практически так же, как с объектами любого другого класса.

Когда хранимый метод Java выполняет команду SQL и при этом происходит исключение, оно представляется объектом subclasses `java.sql.SQLException`. Этот subclass содержит два метода, которые возвращают код ошибки Oracle и сообщение об ошибке: `getErrorCode` и `getMessage`.

Если хранимая процедура Java, вызванная из SQL или PL/SQL, инициирует исключение, которое не перехватывается виртуальной машиной, вызывающая сторона получает инициированное исключение с сообщением об ошибке Java. Так передается информация обо всех перехваченных исключениях (в том числе не относящихся к SQL). Рассмотрим разные способы обработки ошибок и информации, которая при этом выводится.

Допустим, я создаю класс, который использует JDBC для удаления объектов в базе данных (код взят из примера в документации Oracle):

```
/* Файл в Сети: DropAny.java */
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
```

продолжение ➤

```

public class DropAny {
    public static void object (String object_type, String object_name)
        throws SQLException {
        // Подключение к Oracle с использованием драйвера JDBC
        Connection conn = new OracleDriver().defaultConnection();
        // Построение команды SQL
        String sql = "DROP " + object_type + " " + object_name;
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(sql);
        }
        catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        finally {
            stmt.close();
        }
    }
}

```

Этот фрагмент перехватывает и выводит все исключения `SQLException` в выделенном коде. Секция `finally` гарантирует, что метод `close` будет выполнен независимо от того, произошло исключение или нет (это необходимо для правильной обработки открытого курсора).



Хотя использование JDBC для удаления объектов не имеет особого смысла, потому что это гораздо проще делается в динамическом SQL, реализация на Java позволяет пользоваться этой функциональностью из других Java-программ без обращения к PL/SQL.

Класс загружается в базу данных командой `loadjava`, после чего упаковывается в процедуру PL/SQL следующим образом:

```

PROCEDURE dropany (
    tp IN VARCHAR2,
    nm IN VARCHAR2
)
AS LANGUAGE JAVA
    NAME 'DropAny.object (
        java.lang.String,
        java.lang.String)';

```

При попытке удаления несуществующего объекта возможен один из двух результатов:

```

SQL> CONNECT scott/tiger
Connected.

```

```

SQL> SET SERVEROUTPUT ON
SQL> BEGIN dropany ('TABLE', 'blip'); END;/
PL/SQL procedure successfully completed.

```

```

SQL> CALL DBMS_JAVA.SET_OUTPUT (1000000);

Call completed.

```

```

SQL> BEGIN dropany ('TABLE', 'blip'); END;/
ORA-00942: table or view does not exist

```

Эти примеры лишний раз напомним вам, что вывод `System.err.println` не появится на экране до тех пор, пока вы не включите его явно вызовом `DBMS_JAVA.SET_OUTPUT`. Впрочем, в любом случае в вызывающий блок никакое исключение не передается, потому что оно перехватывается в Java. После второго вызова `dropany` мы видим, что сообщение об ошибке, переданное методом `getMessage`, берется прямо из Oracle.

Если закомментировать обработчик исключения в методе `DropAny.object`, результат будет выглядеть примерно так (режим `SERVEROUTPUT` включен — как и вывод Java):

```
SQL > BEGIN
  2   dropany('TABLE', 'blip');
  3   EXCEPTION
  4       WHEN OTHERS
  5       THEN
  6           DBMS_OUTPUT.PUT_LINE(SQLCODE);
  7           DBMS_OUTPUT.PUT_LINE(SQLERRM);
  8   END;
  9   /
oracle.jdbc.driver.OracleSQLException: ORA-00942: table or view does not exist
at oracle.jdbc.driver.T2SConnection.check_error(T2SConnection.java:120)
at oracle.jdbc.driver.T2SStatement.check_error(T2SStatement.java:57)
at oracle.jdbc.driver.T2SStatement.execute_for_rows(T2SStatement.java:486)
at oracle.jdbc.driver.OracleStatement.doExecute
WithTimeout(OracleStatement.java:1148)
at oracle.jdbc.driver.OracleStatement.executeUpdate(OracleStatement.java:1705)
at DropAny.object(DropAny:14)
```

-29532

ORA-29532: Java call terminated by uncaught Java exception: java.sql.SQLException:
ORA-00942: table or view does not exist

Пожалуй, это стоит пояснить. Все, что находится между строками

java.sql.SQLException: ORA-00942: table or view does not exist

и

-29532

представляет дамп стека ошибок; Java генерирует его и отправляет в стандартный вывод независимо от того, как ошибка обрабатывается в PL/SQL. Другими словами, даже если мой раздел исключений будет иметь следующий вид:

```
EXCEPTION WHEN OTHERS THEN NULL;
```

эти данные все равно будут выданы на экран, после чего продолжится выполнение внешнего блока (если он есть). Последние три строки генерируются вызовами `DBMS_OUTPUT.PUT_LINE`. Обратите внимание: вместо кода ошибки Oracle ORA-00942 выводится код ORA-29532, обобщенная ошибка Java. Это создает проблемы — если вы перехватите ошибку, как узнать, что произошло?

Насколько мне удалось выяснить, ошибка возвращается `SQLERRM` в формате

ORA-29532: Java call ...: java.sql.SQLException: ORA-NNNNN ...

Следовательно, я могу поискать вхождение `java.sql.SQLException` и выделить подстроку вызовом `SUBSTR` с этой позиции.

На сайте книги имеется файл `getErrorInfo.sp` с программой, которая возвращает код и сообщение для текущей ошибки. Дополнительная информация компенсирует недостатки формата сообщения об ошибке Java.

В следующих разделах мы расширим класс `JDelete` до класса `JFile`, существенно увеличивающего возможности работы с файлами из PL/SQL. После этого вы узнаете, как писать классы Java и программы PL/SQL на базе этих классов для работы с объектами Oracle.

Расширение функциональности файлового ввода/вывода

Пакет Oracle `UTL_FILE` (см. главу 22) замечателен скорее своими недостатками, нежели достоинствами. Он поддерживает последовательное чтение и запись содержимого файлов... и всё. По крайней мере до выхода Oracle9i Database Release 2 пользователь

не мог удалять файлы, изменять привилегии, копировать файлы, читать содержимое каталогов, задавать пути и т. д.

Java приходит на помощь! В Java существует много разных классов для работы с файлами. Вы уже видели класс `File` и знаете, как легко с его помощью добавить в PL/SQL поддержку удаления файлов.

Сейчас мы создадим новый класс `JFile`, с помощью которого разработчик PL/SQL сможет получить ответы на вопросы и выполнить операции из следующего списка:

- Возможно ли чтение из файла? Запись в файл? Существует ли файл? Является ли заданный объект файлом или каталогом?
- Каков размер файла в байтах? В каком каталоге находится файл?
- Как получить имена всех файлов в каталоге, соответствующих заданному фильтру?
- Как создать каталог? Переименовать файл? Изменить расширение файла?

Я не стану подробно описывать все методы класса `JFile` и соответствующий пакет; в них много повторяющихся фрагментов, а большинство методов Java почти не отличается от функции удаления, построенной в начале этой главы. Вместо этого я сосредоточусь на специфических особенностях разных частей класса и пакета. Полное определение кода содержится в следующих файлах на сайте книги:

- `JFile.java` — класс Java, который объединяет разные виды информации о файлах операционной системы и предоставляет доступ к ней через программный интерфейс, доступный из PL/SQL.
- `xfile.pkg` — пакет PL/SQL, обертка для класса `JFile`.



В Oracle9i Database Release 2 появилась расширенная версия пакета `UTL_FILE`, которая, среди прочего, позволяет удалять файлы процедурой `UTL_FILE.FREMOVE`. Также поддерживаются операции копирования (`FCOPY`) и переименования (`FRENAME`) файлов.

Доработка метода удаления

Прежде чем переходить к новым интересным вопросам, следует убедиться в том, что сделанное ранее работает идеально. Мое определение метода `JDelete.delete` и функции `delete_file` далеко от идеала. Код метода, который уже приводился ранее, выглядит так:

```
public static int delete (String fileName) {
    File myFile = new File (fileName);
    boolean retval = myFile.delete();
    if (retval) return 1; else return 0;
}
```

Связанный с ним код PL/SQL:

```
FUNCTION fDelete (
    file IN VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
    NAME 'JDelete.delete (java.lang.String)
        return int';
```

В чем проблема? В том, что я был вынужден использовать неудобные числовые представления для `TRUE/FALSE`. В результате мне пришлось писать код следующего вида:

```
IF fdelete ('c:\temp\temp.sql') = 1 THEN ...
```

Получается уродливый код с жестко запрограммированными значениями. Кроме того, любой разработчик, пишущий код PL/SQL, должен будет знать о значениях `TRUE` и `FALSE`, встроенных в класс Java.

Функция `delete_file` гораздо лучше смотрелась бы с заголовком следующего вида:

```
FUNCTION fDelete (
  file IN VARCHAR2) RETURN BOOLEAN;
```

Итак, давайте посмотрим, что нужно для того, чтобы предоставить пользователям пакета `xfile` этот простой и удобный интерфейс.

Сначала класс `JDelete` будет переименован в `JFile`, чтобы в имени отражалось расширение его функциональности. Затем я добавлю методы, инкапсулирующие значения `TRUE/FALSE`, возвращаемые другими методами, — и вызову их из метода `delete`. Результат выглядит так:

```
/* Файл в Сети: JFile.java */
import java.io.File;

public class JFile {

  public static int tVal () { return 1; };
  public static int fVal () { return 0; };

  public static int delete (String fileName) {
    File myFile = new File (fileName);
    boolean retval = myFile.delete();
    if (retval) return tVal();
    else return fVal();
  }
}
```

Проблемы на стороне Java решены; переходим к пакету PL/SQL. Первая версия спецификации `xfile`:

```
/* Файл в Сети: xfile.pkg */
PACKAGE xfile
IS
  FUNCTION delete (file IN VARCHAR2)
    RETURN BOOLEAN;
END xfile;
```

Как реализовать функцию, возвращающую логическое значение? Передо мной стоят две задачи:

- Скрыть тот факт, что для передачи `TRUE` или `FALSE` используются числовые значения.
- Избегать жесткого кодирования значений 1 и 0 в пакете.

Для этого я определяю в своем пакете две глобальные переменные для хранения числовых значений:

```
/* Файл в Сети: xfile.pkg */
PACKAGE BODY xfile
IS
  g_true INTEGER;
  g_false INTEGER;
```

В конце тела пакета создается инициализационный раздел, который вызывает эти программы для инициализации глобальных переменных. Выполняя этот шаг в разделе инициализации, я избегаю лишних вызовов методов Java (и связанных с ними затрат):

```
BEGIN
  g_true := tval;
  g_false := fval;
END xfile;
```

В разделе объявлений тела пакета определяются две приватные функции; их единственной целью является доступ из кода PL/SQL к методам `JFile`, инкапсулирующим значения 1 и 0:

```

FUNCTION tval RETURN NUMBER
AS LANGUAGE JAVA
  NAME 'JFile.tVal () return int';
FUNCTION fval RETURN NUMBER
AS LANGUAGE JAVA
  NAME 'JFile.fVal () return int';

```

Чтобы в спецификации пакета можно было использовать функцию с логическим возвращаемым значением, я создаю приватную функцию «внутреннего удаления», которая представляет собой обертку для метода `JFile.delete`. Функция возвращает число:

```

FUNCTION Idelete (file IN VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
  NAME 'JFile.delete (java.lang.String) return int';

```

Наконец, моя открытая функция `delete` теперь может вызвать `Idelete` и преобразовать целое число в логическое значение, проверяя его по глобальной переменной:

```

FUNCTION delete (file IN VARCHAR2) RETURN BOOLEAN
AS
BEGIN
  RETURN Idelete (file) = g_true;
EXCEPTION
  WHEN OTHERS
  THEN
    RETURN FALSE;
END;

```

Так логическое значение Java преобразуется в логическое значение PL/SQL. Этот прием неоднократно применяется в теле пакета `xfile`.

Чтение содержимого каталога

Одна из самых интересных возможностей `JFile` — получение списка файлов в каталоге. Она реализуется вызовом метода `File.list`; если строка, которая использовалась для построения нового объекта `File`, содержит имя каталога, то метод возвращает массив строк с именами файлов, находящихся в этом каталоге. Давайте посмотрим, как предоставить доступ к этой информации в виде коллекции PL/SQL.

Начнем с создания типа коллекции:

```
CREATE OR REPLACE TYPE dirlist_t AS TABLE OF VARCHAR2(512);
```

Затем создается метод `dirlist`, который возвращает `oracle.sql.ARRAY`:

```

/* Файл в Сети: JFile.java */
import java.io.File;
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;
public class JFile {
...

  public static oracle.sql.ARRAY dirlist (String dir)
    throws java.sql.SQLException
  {
    Connection conn = new OracleDriver().defaultConnection();
    ArrayDescriptor arraydesc =
      ArrayDescriptor.createDescriptor ("DIRLIST_T", conn);

    File myDir = new File (dir);
    String[] fileList = myDir.list();

    ARRAY dirArray = new ARRAY(arraydesc, conn, fileList);
    return dirArray;
  }
...

```

Метод сначала получает «дескриптор» пользовательского типа `dirlist_t` для создания соответствующего объекта. После вызова метода `Java File.list` он копирует полученный список файлов в объект `ARRAY` при вызове конструктора.

На стороне PL/SQL создается обертка, которая вызывает этот метод:

```
FUNCTION dirlist (dir IN VARCHAR2)
  RETURN dirlist_t
AS
  LANGUAGE JAVA
  NAME 'myFile.dirlist(java.lang.String) return oracle.sql.ARRAY';
```

Простой пример возможного вызова:

```
DECLARE
  tempdir dirlist_t;
BEGIN
  tempdir := dirlist('C:\temp');
  FOR indx IN 1..tempdir.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE_(tempdir(indx));
  END LOOP;
END;
```

В пакет `xfile` входят дополнительные программы для выполнения других операций: получения имен файлов в виде списка (вместо массива), ограничения списка файлов по заданному фильтру и изменения расширения заданных файлов. Вы найдете в нем все точки входа пакета `UTL_FILE`, такие как `FOPEN` и `PUT_LINE`. Я добавил их, чтобы вы могли избежать использования `UTL_FILE` для любых целей, кроме объявлений файловых дескрипторов в виде `UTL_FILE.FILE_TYPE` и ссылок на исключения, объявленных в `UTL_FILE`.

Другие примеры

На сайте книги имеются другие интересные примеры применения Java для расширения функциональности PL/SQL или выполнения более сложного отображения типов данных:

- `utlzip.sql` — класс Java и соответствующий пакет для использования функциональности `zip`/сжатия в PL/SQL (предоставлен рецензентом Вадимом Лоевским). Также можно воспользоваться командой `CREATE OR REPLACE JAVA` для прямой загрузки класса в базу данных без команды `loadjava`. Заголовок команды создания класса Java:

```
/* Файл в Сети: utlzip.sql */
JAVA SOURCE NAMED "UTLZip" AS
import java.util.zip.*;
import java.io.*;
public class utlzip
{ public static void compressfile(string infilename, string outfilename)
  ...
}
```

А вот обертка для метода Java:

```
PACKAGE utlzip
IS
  PROCEDURE compressfile (p_in_file IN VARCHAR2, p_out_file IN VARCHAR2)
  AS
    LANGUAGE JAVA
    NAME 'UTLZip.compressFile(java.lang.String,
                               java.lang.String)';
END;
```

В Oracle Database 10g и выше вы можете просто воспользоваться встроенным пакетом Oracle `UTL_COMPRESS`.

- `DeleteFile.java` и `deletefile.sql` — класс Java и код PL/SQL, демонстрирующий передачу коллекций (вложенных таблиц или `VARRAY`) в массивы Java (предоставлен рецензен-

том Алексом Романкевичем). Код реализует удаление всех файлов в заданном каталоге, измененных после заданной даты. В реализации на стороне PL/SQL я сначала создаю вложенную таблицу объектов, а затем передаю ее Java при помощи класса `oracle.sql.ARRAY`:

```
CREATE TYPE file_details AS OBJECT (
    dirname      VARCHAR2 (30),
    deletedate    DATE)
/
CREATE TYPE file_table AS TABLE OF file_details;
/
CREATE OR REPLACE PACKAGE delete_files
IS
    FUNCTION fdelete (tbl IN file_table) RETURN NUMBER
    AS
    LANGUAGE JAVA
        NAME 'DeleteFile.delete(oracle.sql.ARRAY) return int';
END delete_files;
```

Ниже приводится начало метода Java (за полной реализацией и подробными комментариями обращайтесь к сценарию `DeleteFile.java`). Алекс извлекает результирующий набор из структуры массива, после чего перебирает элементы этого набора:

```
/* Файлы в Сети: DeleteFile.java and deletefile.sql */
public class DeleteFile {
    public static int delete(oracle.sql.ARRAY tbl) throws SQLException {
        try {
            // Получение содержимого таблицы/varray как результирующего набора
            ResultSet rs = tbl.getResultSet();
            for (int ndx = 0; ndx < tbl.length(); ndx++) {
                rs.next();
                // Получение индекса и элемента массива
                int aryndx = (int)rs.getInt(1);
                STRUCT obj = (STRUCT)rs.getObject(2);
```

- `utlcmd.sql` — класс Java и пакет, при помощи которых можно легко (и даже слишком!) выполнить любую команду операционной системы из PL/SQL. Будьте осторожны!

28

Внешние процедуры

На заре существования PL/SQL часто приходилось слышать вопрос: «А из Oracle можно выполнить произвольный код?» Как правило, речь идет об отправке электронной почты, выполнении команд операционной системы или использовании функциональности, отсутствующей в PL/SQL. Хотя после того, как в поставку Oracle были включены встроенные пакеты `UTL_SMTP` и `UTL_MAIL`, в настоящее время у выполнения произвольного кода существует немало альтернатив. Лишь самые распространенные решения:

- Напишите программу в виде хранимой процедуры Java и вызовите ее из PL/SQL.
- Используйте таблицу базы данных или очередь для хранения запросов. Организуйте чтение и обработку запросов из отдельного процесса.
- Предоставьте доступ к программе как к веб-службе.
- Используйте канал к базе данных и напишите демон, который реагирует на запросы в канале.
- Напишите программу на C и вызовите ее как внешнюю процедуру.

Решение с Java легко реализуется и достаточно быстро работает для многих приложений. Технология очередей весьма интересна, но даже при простом использовании обычных таблиц потребуются два сеанса базы данных Oracle: один для записи в очередь, другой для чтения из нее. Кроме того, наличие двух сеансов означает появление двух разных транзакционных пространств, что может создать проблемы в приложении. Решения на базе каналов также страдают от проблемы двух сеансов, не говоря уже о проблеме упаковки и распаковки содержимого канала. Вдобавок обработка множества одновременных запросов, использующих любые из перечисленных решений, может потребовать создания собственной системы прослушивания и диспетчеризации.

Все эти причины могут заставить серьезно рассмотреть последний вариант. *Внешние процедуры* позволяют PL/SQL сделать практически все, что можно сделать в любом другом языке; они также исправляют недостатки других перечисленных методов. Но как работают внешние процедуры? Насколько они безопасны? Как построить собственную внешнюю процедуру? Каковы их преимущества и недостатки? В этой главе мы получим ответы на эти вопросы, а также рассмотрим типичные примеры ситуаций, в которых используются внешние процедуры.

Кстати говоря, в примерах этой главы используется пакет GCC (GNU Compiler Collection), который, как и Oracle, встречается практически на всех платформах¹. Мне нравится GCC, но возможно, вам придется использовать компилятор, «родной» для вашего оборудования.

¹ GCC распространяется бесплатно; для Microsoft Windows существуют по крайней мере две версии. В этой главе используется версия MinGW.

Знакомство с внешними процедурами

Чтобы вызвать внешнюю программу из Oracle, программа должна выполняться как общая библиотека. В операционных системах Microsoft программы такого рода называются *DLL* (Dynamically Linked Library); в Solaris, AIX и Linux файлы общих библиотек обычно имеют расширение *.so* (Shared Object) или *.sl* (Shared Library) в HP-UX. Теоретически внешнюю процедуру можно написать на любом языке, но компилятор и компоновщик должны уметь генерировать общие библиотеки в формате, допускающем вызов из C. Внешняя программа «публикуется» при помощи специальной обертки PL/SQL, называемой *спецификацией вызова*. Если внешняя функция возвращает значение, она связывается с функцией PL/SQL; если внешняя функция не возвращает ничего, она связывается с процедурой PL/SQL.

Пример: вызов команды операционной системы

Наш первый пример позволяет выполнить из программы любую команду уровня операционной системы. Что-что? Наверняка у вас сработала внутренняя сигнализация — ведь такие вещи опасны, не так ли? Несмотря на все препоны безопасности, которые вам придется обойти, администратор базы данных все равно не захочет предоставлять широкие разрешения для выполнения этого кода. Впрочем, придержите свое недоверие, пока мы будем рассматривать примеры.

Первый пример состоит из очень простой функции C `extprocsh`, которая получает строку и передает ее функции `system` для выполнения:

```
int extprocshell(char *cmd)
{
    return system(cmd);
}
```

Функция возвращает код результата, полученный от `system` — функции, обычно находящейся в библиотеке времени выполнения C (`libc`) в Unix, или в библиотеке `msvcrt.dll` на платформах Microsoft.

Сохранив исходный код в файле с именем `extprocsh.c`, я использую компилятор GNU C для построения общей библиотеки. На моей 64-разрядной машине с системой Solaris, на которой работает GCC 3.4.2 и Oracle Database 10g Release 2, я использовал следующую команду компилятора (обратите внимание: параметры GCC зависят от дистрибутива Unix/Linux):

```
gcc -m64 extprocsh.c -fPIC -G -o extprocsh.so
```

На машине с Microsoft Windows XP Pro, на которой работает running GCC 3.2.3 из Minimal GNU for Windows (MinGW) и Oracle Database 10g Release 2, следующая команда работает:

```
c:\MinGW\bin\gcc extprocsh.c -shared -o extprocsh.dll
```

Эти команды создают файл общей библиотеки `extprocsh.so` или `extprocsh.dll`. Теперь нужно разместить библиотеку там, где Oracle сможет ее найти. В зависимости от версии Oracle это может быть легче сказать, чем сделать! В табл. 28.1 приводятся некоторые рекомендации по размещению файлов.

После копирования файла и/или внесения изменений в конфигурацию также необходимо определить в Oracle ссылку на DLL:

```
CREATE OR REPLACE LIBRARY extprocshell_lib
AS '/u01/app/oracle/local/lib/extprocsh.so'; -- Unix/Linux
```



```
CREATE OR REPLACE LIBRARY extprocshell_lib
AS 'c:\oracle\local\lib\extprocsh.dll';      -- Microsoft
```

Таблица 28.1. Местонахождение файлов общих библиотек

| Версия | Стандартное местонахождение | Возможность определения нестандартного местонахождения |
|---|---|---|
| Oracle8 Database, Oracle8i Database, Oracle9i Database Release 1 | Любой каталог, доступный для чтения процессу Oracle | Отсутствует |
| Oracle9i Release 2 и выше | \$ORACLE_HOME/lib и/или \$ORACLE_HOME/bin (в зависимости от версии и платформы Oracle; В Unix обычно используется lib, в Microsoft Windows — bin) | Редактирование файла конфигурации слушателей и определение пути для свойства ENV\$="EXTPROC_DLLS..." (см. раздел «Конфигурация Oracle Net») |

Пусть вас не вводит в заблуждение термин «библиотека» (library); в действительности это обычный псевдоним для имени файла, который может использоваться в пространстве имен Oracle. Также обратите внимание на то, что для выполнения этого шага необходима привилегия Oracle CREATE LIBRARY (одна из тех препон из области безопасности, о которых упоминалось выше).

Теперь я могу создать спецификацию вызова PL/SQL, использующую только что созданную библиотеку:

```
FUNCTION shell(cmd IN VARCHAR2)
RETURN PLS_INTEGER
AS
LANGUAGE C
LIBRARY extprocshell_lib
NAME "extprocshell"
PARAMETERS (cmd STRING, RETURN INT);
```

Вот и все! Если администратор базы данных настроил поддержку внешних процедур в системной среде (см. раздел «Определение конфигурации слушателей»), функция shell теперь доступна всюду, где могут вызываться функции PL/SQL — SQL*Plus, Perl, Pro*C и т. д. С точки зрения прикладного программирования вызов внешней процедуры неотличим от вызова традиционной процедуры, например:

```
DECLARE
result PLS_INTEGER;
BEGIN
result := shell('команда');
END;
```

Или даже так:

```
SQL> SELECT shell('команда') FROM DUAL;
```

При успешном выполнении возвращается ноль:

```
SHELL('команда')
-----
0
```

Учтите, что команда операционной системы обычно направляет данные в стандартный поток вывода или ошибок, и этот вывод просто пропадет, если не изменить программу для его передачи в PL/SQL. Вы можете перенаправить его в файл (с учетом разрешений уровня операционной системы); простейший пример сохранения списка содержимого каталога в файле:

```
result := shell('ls / > /tmp/extproc.out');      -- Unix/Linux
result := shell('cmd /c "dir c:\ > c:\temp\extproc.out"'); -- Microsoft
```

Эти команды операционной системы выполняются с теми же привилегиями, что и слушатель Oracle Net, запустивший процесс `extproc`. Хмм, наверняка ваш администратор или специалист по безопасности с этим не смирится... Читайте дальше, если вы хотите ему помочь.

Архитектура внешних процедур

Что происходит «внутри» при вызове внешней процедуры? Начнем с ситуации, представленной в предыдущем разделе, использующей «агента» внешних процедур по умолчанию.

Когда исполнительное ядро PL/SQL узнает из откомпилированного кода, что программа имеет внешнюю реализацию, оно обращается к службе TNS с именем `EXTPROC_CONNECTION_DATA`, информация о которой должна быть предоставлена серверу через один из механизмов имен Oracle — например, файл `tnsnames.ora`. Как показано на рис. 28.1, слушатель Oracle Net отвечает на запрос, порождая сеансовый процесс с именем `extproc`, которому передается путь к файлу DLL с именем функции и аргументами. Именно процесс `extproc` осуществляет динамическую загрузку общей библиотеки, передает необходимые аргументы, получает вывод и возвращает результаты вызывающей стороне. В этой схеме для каждого сеанса Oracle выполняется только процесс `extproc`; он запускается с вызовом первой внешней процедуры и завершается с отключением сеанса. Для каждого вызова новой внешней процедуры `extproc` загружает соответствующую общую библиотеку (если она не была загружена ранее).

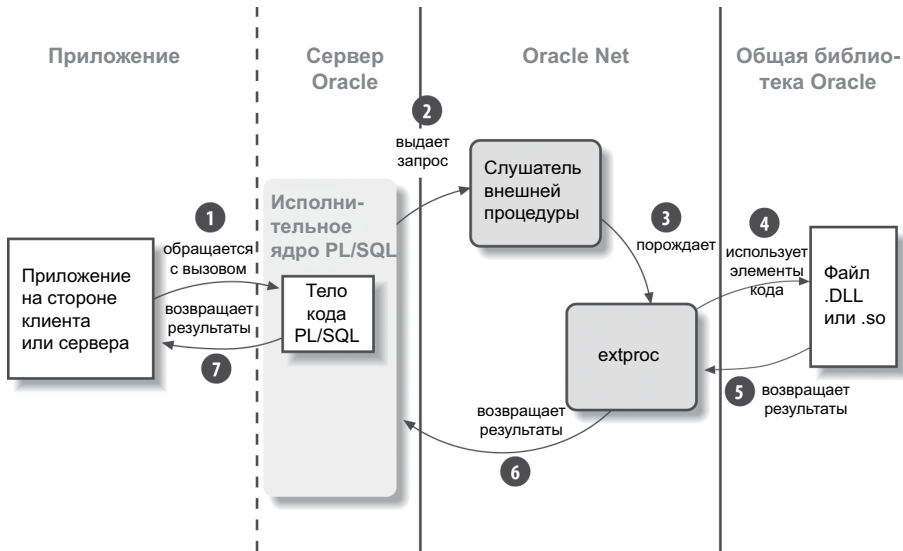


Рис. 28.1. Вызов внешней процедуры с использованием агента по умолчанию

В Oracle реализованы некоторые возможности, которые делают внешние процедуры эффективным и практичным инструментом программирования.

- **Общие DLL-библиотеки** — внешняя программа `C` должна быть общей динамически скомпонованной библиотекой (вместо модуля со статической компоновкой). Схема с компоновкой, отложенной до стадии выполнения, сопряжена с некоторыми дополнительными затратами, но использование общей библиотеки в нескольких сеансах обеспечивает экономию памяти; операционная система обеспечивает совместный

доступ к страницам памяти библиотеки со стороны нескольких процессов. Другое преимущество модулей с динамической компоновкой — простота создания и обновления модулей по сравнению со статической компоновкой. Кроме того, общая библиотека может содержать много подпрограмм (отсюда и термин «библиотека»), а это сокращает затраты, поскольку системе приходится динамически загружать меньшее количество файлов.

- **Раздельное пространство памяти** — внешние процедуры Oracle выполняются в пространстве памяти, отдельном от пространства процессов ядра базы данных. Если во внешней процедуре происходит сбой, он не затрагивает памяти ядра; процесс `extproc` просто возвращает ошибку ядру PL/SQL, которое в свою очередь передает информацию о ней приложению. Написать внешнюю процедуру, которая будет вызывать сбой сервера Oracle, возможно, но сделать это будет не проще, чем из не-внешней процедуры.
- **Полная поддержка транзакций** — внешние процедуры предоставляют полную поддержку транзакций; иначе говоря, они могут полноценно участвовать в текущей транзакции. Получая «контекстную» информацию от PL/SQL, процедура может обращаться к базе данных за данными, выдавать вызовы SQL или PL/SQL и инициализировать исключения. Использование этих функций требует низкоуровневого программирования OCI (Oracle Call Interface)... но по крайней мере это возможно!
- **Многопоточность** (Oracle Database 10g и выше) — в Oracle9i Database для каждого сеанса Oracle, вызывавшего внешнюю процедуру, требовался вспомогательный процесс `extproc`. При большом количестве пользователей лишние затраты могли быть довольно значительными. Начиная с Oracle Database 10g администратор может настроить многопоточного «агента», который обслуживает каждый запрос в программном потоке, а не в отдельном процессе. Если вы захотите пойти этим путем, убедитесь в том, что ваша программа С безопасна по отношению к потокам. За дополнительной информацией обращайтесь к разделу «Настройка многопоточного режима».

Несмотря на все свои преимущества, внешние процедуры не идеальны: архитектура Oracle неизбежно требует значительных межпроцессных взаимодействий. Эти затраты компенсируются безопасностью отделения пространства памяти внешней процедуры от пространства памяти сервера базы данных.

Конфигурация Oracle Net

Рассмотрим процесс создания простой конфигурации, которая поддерживает вызов внешних процедур с закрытием некоторых зияющих брешей в области безопасности.

Определение конфигурации слушателей

Связь PL/SQL с общими библиотеками устанавливается на коммуникационном уровне Oracle Net. И хотя в установке по умолчанию Oracle8i Database и выше обычно предоставляется поддержка внешних процедур, скорее всего, вам не придется использовать стандартную конфигурацию, пока Oracle не внесет значительные улучшения в области безопасности.

На момент написания третьего издания книги репутация Oracle была изрядно подорвана уязвимостью, обусловленной вызовом внешних процедур. А именно удаленный атакующий мог подключиться к порту TCP/IP Oracle NET (обычно 1521) и запустить `extproc` без аутентификации. И хотя эта конкретная уязвимость была закрыта, здравый смысл в области безопасности Oracle требует всегда выполнять правило в следующей врезке.



Слушатели Oracle должны находиться за брандмауэром; никогда не предоставляйте доступ к порту слушателя через Интернет или другую сеть без доверия.

Правильная настройка слушателя требует изменения файла `tnsnames.ora` и `listener.ora` (либо вручную, либо через интерфейсную часть Oracle Net Manager). Ниже приведен простой файл `listener.ora`, который настраивает слушателя внешней процедуры отдельно от слушателя базы данных:

```
### Обычный слушатель (для подключения к базе данных)
LISTENER =
  (ADDRESS = (PROTOCOL = TCP)(HOST = хост)(PORT = 1521))

SID_LIST_LISTENER =
  (SID_DESC =
    (GLOBAL_DBNAME = глобальное_имя)
    (ORACLE_HOME = домашний_каталог_oracle)
    (SID_NAME = SID)
  )

### Слушатель внешней процедуры
EXTPROC_LISTENER =
  (ADDRESS = (PROTOCOL = IPC)(KEY = ключ))

SID_LIST_EXTPROC_LISTENER =
  (SID_DESC =
    (SID_NAME = SID_внешней_процедуры)
    (ORACLE_HOME = домашний_каталог_oracle)
    (ENVS="EXTPROC_DLLS=список_файлов,другие_переменные_окружения")
    (PROGRAM = extproc)
  )
```

В этом фрагменте:

Хост и *глобальное_имя* — имя или IP-адрес машины и полное имя базы данных соответственно. В приведенном примере эти параметры относятся только к слушателю базы данных, но не к слушателю внешней процедуры.

Ключ — короткий идентификатор, по которому Oracle Net отличает этого слушателя от других возможных слушателей межпроцессных взаимодействий (IPC). Имя выбирается произвольно, потому что программы его все равно не видят. Oracle использует идентификатор `EXTPROC0` или `EXTPROC1` как имя по умолчанию для первой установки Oracle Net на заданной машине. Этот идентификатор должен совпадать в списках адресов `listener.ora` и `tnsnames.ora`.

Домашний_каталог_oracle — полный путь `ORACLE_HOME`; например `/u01/app/oracle/oracle/product/10.2.0/db_1` в Unix или `C:\oracle\product\10.2.0\db_1` в Microsoft Windows. Обратите внимание: имя каталога не заключается в кавычки и не завершается косой чертой.

SID_внешней_процедуры — произвольный уникальный идентификатор слушателя внешней процедуры. В установке по умолчанию Oracle использует значение `PLSExtProc`. `ENVS="EXTPROC_DLLS=список_файлов"` — секция `ENVS`, доступная в Oracle9i Database Release 2 и выше, задает переменные окружения для слушателя. Элементы списка разделяются двоеточиями, при этом каждый элемент должен содержать полный путь к файлу общего объекта.

В списке могут использоваться специальные ключевые слова. Ключевое слово `ANY` представляет минимальный уровень безопасности; с ним можно использовать любую общую библиотеку, видимую пользователю операционной системы, запускающему слушателя внешней процедуры (`ENVS="EXTPROC_DLLS=ANY"`).

Ключевое слово **ONLY**, соответствующее максимальному уровню безопасности, разрешает выполнение только общих библиотек из списка, разделенного двоеточиями. На моей машине с системой Solaris он может выглядеть так:

```
(ENVS="EXTPROC_DLLS=ONLY:/u01/app/oracle/local/lib/extprocsh.so:/u01/app/oracle/local/lib/RawdataToPrinter.so")
```

А вот аналогичная запись с ноутбука с операционной системой Microsoft Windows:

```
(ENVS="EXTPROC_DLLS=ONLY:c:\oracle\admin\local\lib\extprocsh.dll;c:\oracle\admin\local\lib\RawDataToPrinter.dll")
```

Здесь двоеточие (:) используется в двух разных контекстах: как разделитель списка и как обозначение буквы диска. Я указал всего два библиотечных файла, но вы можете включить столько, сколько потребуется.

Если опустить ключевые слова **ANY** и **ONLY**, но передать список файлов, будут доступны как каталоги по умолчанию, так и явно указанные файлы.

Другие_переменные_окружения — чтобы задать значения переменных окружения, используемых общими библиотеками, добавьте их в секцию **ENVS** слушателя внешней процедуры. В Unix часто используется значение **LD_LIBRARY_PATH**:

```
(ENVS="EXTPROC_DLLS=список_файлов_общих_объектов,LD_LIBRARY_PATH=/usr/local/lib")
```

Элементы списка файлов и переменные окружения разделяются запятыми.

Характеристики безопасности

Приведенная конфигурация решает две важные задачи из области безопасности:

- Она позволяет системному администратору запускать слушателей внешних процедур от имени учетной записи с ограниченными привилегиями. По умолчанию слушатель выполняется с правами учетной записи, запустившей сервер Oracle.
- Слушатель внешней процедуры может принимать только подключения IPC с локальной машины, но не подключения TCP/IP с других машин.

Впрочем, это еще не все. Файл **tnsnames.ora** базы данных, из которого взят этот фрагмент, должен содержать запись следующего вида:

```
EXTPROC_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = IPC)(KEY = extprocKey))
    (CONNECT_DATA = (SID = extprocSID) (PRESENTATION = RO))
  )
```

Большая часть настроек должна быть знакома по конфигурации слушателя, описанной ранее. Обратите внимание: значения *ключ* и *SID_внешней_процедуры*, использованные для слушателя, должны совпадать с соответствующими значениями в этом файле. Необязательный параметр **PRESENTATION** немного повышает производительность; он сообщает серверу, который может вести прослушивание по нескольким протоколам, что клиент желает взаимодействие по протоколу Remote-Ops (отсюда и сокращение **RO**).

Будьте осторожны с привилегиями вспомогательной учетной записи слушателя — особенно в отношении изменения файлов, принадлежащих операционной системе или учетной записи **oracle**. Кроме того, задавая переменную окружения **TNS_ADMIN** в Unix (или в реестре Windows), вы можете переместить файлы **listener.ora** и **sqlnet.ora** слушателей внешних процедур в отдельный каталог. Это может стать еще одним аспектом общей стратегии безопасности.

Может показаться, что настройка файлов конфигурации и создание вспомогательных учетных записей весьма далеки от повседневного программирования PL/SQL, но в наши дни безопасность касается каждого!



Каждый профессионал Oracle должен время от времени посещать страницу оповещений безопасности Oracle. Проблема, о которой я упоминал в разделе «Определение конфигурации слушателей», впервые была опубликована на этой странице в оповещении 29. Регулярно просматривайте этот список — в нем можно найти полезные описания обходных решений или исправлений существующих проблем.

Настройка многопоточного режима

В Oracle Database 10g появилась возможность совместного использования одного процесса внешней процедуры несколькими сеансами. И хотя настройка этого режима требует определенных усилий, эти усилия окупятся при выполнении внешних процедур несколькими пользователями. Минимальные действия, которые должен выполнить администратор для включения многопоточного режима:

1. Остановите слушателя внешней процедуры. Если вы настроили отдельного слушателя так, как было рекомендовано, этот шаг выполняется просто:
2. Отредактируйте файл `listener.ora`: сначала замените ключ внешней процедуры (по умолчанию `EXTPROC0` или `EXTPROC1`) на `PNPKEY`; затем, чтобы полностью исключить всех специализированных слушателей, удалите весь раздел `SID_LIST_EXTPROC_LISTENER`.
3. Отредактируйте файл `tnsnames.ora` и замените ключ на `PNPKEY`.
4. Перезапустите слушателя внешней процедуры, например:

```
OS> lsnrctl stop extproc_listener
```

5. В командной строке ОС задайте значение переменной окружения `AGTCTL_ADMIN`. Значение должно определять полный путь к каталогу; оно сообщает `agtctl`, где должны храниться настройки. (Если переменная `AGTCTL_ADMIN` не задана, но задано значение `TNS_ADMIN`, оно будет использовано автоматически.)
6. Если вам потребуется передать агенту переменные окружения (например, `EXTPROC_DLLS` или `LD_LIBRARY_PATH`), задайте их в текущем сеансе операционной системы. Несколько примеров (для командного интерпретатора `bash` или его эквивалента):

```
OS> export EXTPROC_DLLS=ANY
OS> export LD_LIBRARY_PATH=/lib:/usr/local/lib/sparcv9
```

7. Если вы используете для внешней процедуры `SID` слушателя по умолчанию (то есть `PLSExtProc`), выполните следующую команду:

```
OS> agtctl startup extproc PLSExtProc
```

Чтобы проверить, успешно ли прошла настройка, можно воспользоваться командой `lsnrctl services`:

```
OS> lsnrctl services extproc_listener
...
Connecting to (ADDRESS=(PROTOCOL=IPC)(KEY=PNPKEY))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status READY, has 1 handler(s) for this service...
    Handler(s):
      "ORACLE SERVER" established:0 refused:0 current:0 max:5 state:ready
        PLSExtProc
        (ADDRESS=(PROTOCOL=ipc)(KEY=#5746.1.4))
```

То что нужно — агент находится в состоянии «ready» и не помечен для монопольного использования. Команда также выводит статистику по количеству сеансов; в предыдущем выводе все показатели равны 0, кроме максимального количества сеансов (5 по умолчанию).

Во внутреннем представлении многопоточный агент использует особую схему со слушателями/диспетчерами/рабочими потоками, в которой каждый запрос может передаваться своему потоку выполнения. Для управления количеством задач используются команды «`agtctl set`». Например, чтобы изменить максимальное количество сеансов, сначала остановите агента:

```
OS> agtctl shutdown PLSExtProc
```

Затем задайте значение `max_sessions`:

```
OS> agtctl set max_sessions n PLSExtProc
```

где *n* — максимальное количество сеансов Oracle, который могут одновременно подключаться к агенту.

Наконец, перезапустите агента:

```
OS> agtctl startup extproc PLSExtProc
```

При настройке конфигурации следует знать о некоторых параметрах конфигурации, перечисленных в следующей таблице.

| Параметр | Описание | Значение по умолчанию |
|-------------------------------|--|---|
| <code>max_dispatchers</code> | Максимальное количество потоков-диспетчеров, передающих запросы рабочим потокам | 1 |
| <code>max_task_threads</code> | Максимальное количество рабочих потоков | 2 |
| <code>max_sessions</code> | Максимальное количество сеансов Oracle, которые могут обслуживаться многопоточным процессом <code>extproc</code> | 5 |
| <code>listener_address</code> | Адреса, с которыми многопоточный процесс регистрируется с уже работающим слушателем | (ADDRESS_LIST= (ADDRESS= (PROTOCOL=IPC) (KEY=PNPKEY)) (ADDRESS= (PROTOCOL=IPC) (KEY=SID_слушателя)) (ADDRESS= (PROTOCOL=TCP) (HOST=127.0.0.1) (PORT=1521))) |

Возможно, в ходе оптимизации параметров агента для разного количества агентских процессов придется поэкспериментировать. Хотя моих экспериментов с многопоточными агентами недостаточно для того, чтобы давать какие-то общие рекомендации, давайте хотя бы рассмотрим изменения, необходимые для использования двух многопоточных агентов. Выполните действия, описанные выше, — но на этот раз на шаге 7 запустите двух агентов с разными именами:

```
OS> agtctl startup extproc PLSExtProc_001
```

```
...
```

```
OS> agtctl startup extproc PLSExtProc_002
```

Также внесите изменения в файл `tnsnames.ora` (шаг 3), чтобы передать Oracle информацию об именах новых агентов.

Чтобы нагрузка распределялась между агентами средствами Oracle Net, отредактируйте раздел `EXTPROC_CONNECTION_DATA` в файле `tnsnames.ora`:

```
EXTPROC_CONNECTION_DATA =
  (DESCRIPTION_LIST =
    (LOAD_BALANCE = TRUE)
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = IPC)(KEY = PNPKEY))
      (CONNECT_DATA = (SID = PLSExtProc_001)(PRESENTATION = RO))
    )
  )
```

продолжение ➤

```

    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = ipc)(key = PNPKEY))
      (CONNECT_DATA = (SID = PLSExtProc_002)(PRESENTATION = RO))
    )
  )

```

Добавьте параметр `DESCRIPTION_LIST` и включите один раздел описания для каждого агента. В новой конфигурации каждый раз, когда Oracle Net получает от PL/SQL запрос на подключение к внешней процедуре, для подключения будет случайно выбран один из перечисленных агентов; если подключение завершится неудачей (например, из-за достижения максимального количества подключаемых сеансов), Oracle Net проверяет другого агента — пока не будет установлено успешное подключение или попытки подключения ко всем агентам завершатся неудачей. В последнем случае будет выдана ошибка ORA-28575 (не удастся открыть подключение RPC к агенту внешней процедуры).

Многопоточный режим и `agtctl` более подробно рассматриваются в руководстве *Oracle Application Development Guide — Fundamentals and the Heterogeneous Connectivity Administrator's Guide*. Средства распределения нагрузки Oracle Net описаны в руководствах *Net Services Administrator's Guide* и *Net Services Reference*.

И последнее замечание: при использовании многопоточных агентов программа C, реализующая внешнюю процедуру, должна быть безопасной по отношению к потокам. Написать такую программу порой бывает очень непросто. Некоторые часто встречающиеся проблемы описаны в конце этой главы.

Создание библиотеки Oracle

Команда SQL `CREATE LIBRARY` определяет в словаре данных Oracle псевдоним для файла внешней общей библиотеки, позволяющий исполнителю ядра PL/SQL найти библиотеку при вызове. Создавать библиотеки могут только администраторы и пользователи, которым были предоставлены привилегии `CREATE LIBRARY` или `CREATE ANY LIBRARY`.

Общий синтаксис команды `CREATE LIBRARY` выглядит так:

```

CREATE [ OR REPLACE ] LIBRARY имя_библиотеки
AS
  'путь_к_файлу' [ AGENT 'агент' ] ;

```

В этом фрагменте:

Имя_библиотеки — действительный идентификатор PL/SQL. Это имя будет использоваться в телах внешних процедур, которым потребуется обратиться с вызовами к общему объекту (или файлу DLL). Имя библиотеки не может совпадать с именем таблицы, объекта PL/SQL верхнего уровня или любого другого объекта в основном пространстве имен.

Путь_к_файлу — полное имя файла общего объекта (или DLL), заключенное в одинарные кавычки.

Начиная с Oracle9i Database в путь к файлу могут включаться переменные окружения. В частности, если учетная запись уровня операционной системы устанавливает значение переменной перед запуском слушателя, вы можете поместить эту переменную в команду `CREATE LIBRARY`, например:

```

CREATE LIBRARY extprocshell_lib AS '${ORACLE_HOME}/lib/extprocsh.so'; -- Unix
CREATE LIBRARY extprocshell_lib AS '%{ORACLE_HOME}%\bin\extprocsh.dll'; -- MS

```

Включение переменных может сделать сценарии более универсальными. Можно использовать переменную окружения, передаваемую в `EXTPROC_DLLS` в файле `listener.ora` (см. выше).

AGENT '*агент*' — канал к базе данных, доступный для владельца библиотеки. Убедитесь в том, что файл `tnsnames.ora` содержит запись для имени службы, указанного при создании канала, и что эта запись включает адрес внешней процедуры и данные подключения. Использование секции AGENT позволяет выполнять внешнюю процедуру на другом сервере базы данных (хотя при этом она должна находиться на том же компьютере). Секция AGENT поддерживается в версиях Oracle9i и выше.

При использовании команды `CREATE LIBRARY` следует помнить о некоторых обстоятельствах:

- Команда должна выполняться администратором базы данных или пользователем, которому были предоставлены привилегии `CREATE LIBRARY` или `CREATE ANY LIBRARY`.
- Библиотеки, как и большинство других объектов баз данных, принадлежат конкретному пользователю Oracle (схеме). Владелец автоматически обладает привилегиями выполнения, он может предоставлять и отзывать привилегию `EXECUTE` для этой библиотеки у других пользователей.
- Другие пользователи, получившие привилегию `EXECUTE` для библиотеки, могут ссылаться на нее в своих спецификациях вызова при помощи синтаксиса *владелец.библиотека*. Также при желании они могут создавать и использовать синонимы для библиотеки.
- Oracle не проверяет существование файла общей библиотеки с указанным именем при выполнении команды `CREATE LIBRARY`. Проверка также не выполняется и при последующем создании объявления внешней процедуры для функции из этой библиотеки. Если путь содержит ошибку, вы узнаете об этом при первой попытке выполнить функцию.

Для каждого используемого файла общей библиотеки достаточно создать только одну библиотеку Oracle. Файл библиотеки может содержать произвольное количество функций C, и ссылки на библиотеку могут присутствовать в любом количестве спецификаций вызова.

А теперь давайте посмотрим, как написать подпрограмму PL/SQL, которая переводит нужную функцию из общей библиотеки в форму, пригодную для вызова из PL/SQL.

Написание спецификации вызова

Внешняя процедура может служить реализацией любой программной единицы, кроме анонимного блока. Другими словами, спецификация вызова может присутствовать в процедуре или функции верхнего уровня, пакетной процедуре или функции или методе объекта. Более того, спецификация вызова может определяться как в спецификации, так и в теле пакетных программных единиц (а также в спецификации или теле объектных типов). Несколько обобщенных примеров:

```
CREATE FUNCTION имя (аргументы) RETURN тип_данных
AS спецификация_вызова;
```

Вероятно, эта форма уже знакома вам по функции `shell`, приведенной ранее в этой главе. Также можно создать процедуру:

```
CREATE PROCEDURE имя
AS спецификация_вызова;
```

В этом случае соответствующая функция C будет возвращать `void`.

Пакетная функция, не требующая тела пакета:

```
CREATE PACKAGE пакет
AS
    FUNCTION имя RETURN тип_данных
    AS спецификация_вызова;
END;
```

Впрочем, когда придет время изменить пакет, спецификацию придется перекомпилировать. В зависимости от вносимых изменений каскадный эффект перекомпиляции можно существенно сократить, переместив спецификацию вызова в тело пакета:

```
CREATE PACKAGE пакет
AS
    PROCEDURE имя;
END;

CREATE PACKAGE BODY пакет
AS
    PROCEDURE имя
    AS спецификация_вызова;
END;
```

Неопубликованные или приватные программные единицы в пакетах также могут быть реализованы в виде внешних процедур. Наконец, использование спецификации вызова в методе объектного типа имеет много общего с ее использованием в пакете; другими словами, спецификацию вызова можно поместить в спецификацию объектного типа или тело соответствующего типа.

Общий синтаксис спецификации вызова

Одним из самых существенных отличий спецификации вызова от обычной хранимой программы является секция `AS LANGUAGE`¹.

Синтаксис этой секции выглядит так:

```
AS LANGUAGE C
    LIBRARY имя_библиотеки
    [ NAME имя_внешней_функции ]
    [ WITH CONTEXT ]
    [ AGENT IN (имя_формального_параметра) ]
    [ PARAMETERS (карта_внешних_параметров) ] ;
```

В этом фрагменте:

- `AS LANGUAGE C` — также может использоваться конструкция `AS LANGUAGE JAVA` (см. главу 27). Другие языки не поддерживаются.
- *имя_библиотеки* — определяемое в команде `CREATE LIBRARY` имя библиотеки, для которой вы имеете привилегию выполнения (либо как владелец, либо в результате получения этой привилегии).
- *имя_внешней_функции* — имя функции, определенное в библиотеке языка C. Если имя записано в нижнем или в смешанном регистре символов, заключите его в двойные кавычки. Этот параметр можно опустить; в этом случае имя внешней функции должно соответствовать имени модуля PL/SQL (по умолчанию используется запись в верхнем регистре).
- `WITH CONTEXT` — указывает, что из PL/SQL вызывающей программе должен передаваться «указатель на контекст». Вызывающая программа должна интерпретировать его как параметр типа `OCIExtProcContext *` (определяемый в заголовочном файле `C ociextp.h`).
- «Контекст», передаваемый через указатель, представляет собой непрозрачную структуру данных, содержащую информацию о сеансе Oracle. Вызываемой процедуре не нужно работать с содержимым структуры данных напрямую; структура просто упрощает другие вызовы OCI, выполняющие специфические операции Oracle — инициирование предопределенных или пользовательских исключений, выделение

¹ В Oracle8 Database эта секция не поддерживается. Вместо нее используется форма `AS EXTERNAL`, которая сейчас считается устаревшей.

памяти уровня сеанса (которая освобождается сразу же после возврата управления PL/SQL) и получение информации об окружении пользователя Oracle.

- **AGENT IN** (*имя формального параметра*) — с библиотекой связывается агентский процесс (по аналогии с секцией **AGENT**), но выбор агента откладывается до стадии выполнения. Значение агента передается как формальный параметр PL/SQL спецификации вызова; оно заменяет имя агента, заданное в библиотеке (если оно есть). За дополнительной информацией о синтаксисе **AGENT IN** обращайтесь к разделу «Нестандартные агенты» этой главы.
- **PARAMETERS** (*карта внешних параметров*) — задает позицию и типы данных параметров, в которых передается информация между PL/SQL и С. *Карта внешних параметров* представляет собой список элементов, разделенных запятыми; эти элементы связываются с параметрами функции С позиционным соответствием или предоставляют дополнительную информацию.

Пожалуй, правильное отображение параметров будет самой сложной из задач, с которыми вы столкнетесь, поэтому в следующем разделе эта задача будет рассматриваться более подробно.

Снова об отображении параметров

Вернемся к проблеме передачи данных между PL/SQL и С. PL/SQL использует собственный набор типов данных, которые только внешне похожи на типы языка С. Переменные PL/SQL могут содержать NULL, на них распространяется тройственная логика таблиц истинности; у переменных С эквивалентной концепции не существует. Библиотека С может не знать, какой набор символов национальных языков используется для выражения алфавитно-цифровых значений. И как функция С должна получать те или иные аргументы — по значению или по ссылке (как указатель)?

Я хочу начать с примера, который расширяет программу `shell`, представленную ранее в этой главе. В последней версии функция `shell` не защищена от вызова с аргументом NULL (вместо нормальной команды). В таком случае вызов `shell(NULL)` приводит к ошибке времени выполнения ORA-01405 (значение выбранного столбца равно NULL). В некоторых приложениях такое поведение может быть вполне допустимым — но что, если вы хотите, чтобы на входное значение NULL внешняя процедура возвращала NULL?

Для нормального выявления Oracle-значения NULL в С требует передачи в PL/SQL дополнительного параметра, называемого *индикатором*. Аналогичным образом, чтобы программа С возвращала Oracle-значение NULL, она должна возвращать в PL/SQL отдельный параметр-индикатор. Хотя Oracle автоматически задает и интерпретирует это значение на стороне PL/SQL, приложение С должно читать и задавать его явно.

Вероятно, лучше всего продемонстрировать эту ситуацию на примере изменения спецификации вызова PL/SQL:

```
FUNCTION shell(cmd IN VARCHAR2)
  RETURN PLS_INTEGER
AS
  LANGUAGE C
  LIBRARY extprocshell_lib
  NAME "extprocsh"
  PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR, RETURN INT);
```

Хотя формальные параметры функции PL/SQL могут находиться где угодно в отображении **PARAMETERS**, элементы отображения должны соответствовать параметрам функции С по позиции и типу данных. Отображение **RETURN** должно быть последним элементом в списке.

Условие RETURN можно исключить из карты параметров, если вы хотите использовать отображение по умолчанию (см. далее). В данном случае это нормально, хотя индикатор все равно должен присутствовать:

```
FUNCTION shell(cmd IN VARCHAR2)
  RETURN PLS_INTEGER
AS
  LANGUAGE C
  LIBRARY extprocshell_lib
  NAME "extprocsh"
  PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR);
```

Хотя спецификация вызова изменилась по сравнению с версией, приведенной ранее в этой главе, программа, вызывающая функцию shell, не видит изменений в количестве или типе данных своих параметров.

Теперь обратимся к новой версии программы C, которая добавляет два параметра, по одному для каждого индикатора:

```
1  #include <ociextp.h>
2
3  int extprocsh(char *cmd, short cmdInd, short *retInd)
4  {
5      if (cmdInd == OCI_IND_NOTNULL)
6      {
7          *retInd = (short)OCI_IND_NOTNULL;
8          return system(cmd);
9      } else
10     {
11         *retInd = (short)OCI_IND_NULL;
12         return 0;
13     }
14 }
```

В следующей таблице кратко описаны основные изменения.

| Строки | Изменения |
|--------|---|
| 1 | На платформах Microsoft включаемый файл находится в подкаталоге %ORACLE_HOME%\oci\include; на машинах Unix я нашел его в каталогах \$ORACLE_HOME/rdbms/demo (Oracle9i Database) и \$ORACLE_HOME/rdbms/public (Oracle Database 10g), хотя он может находиться в любом месте вашей системы |
| 3 | Обратите внимание: индикатор команды относится к типу short, а индикатор возвращаемого значения — short *. Тем самым соблюдается правило передачи аргументов: входные параметры передаются из PL/SQL в C по значению, а выходные и возвращаемые параметры, передаваемые из C в PL/SQL, передаются по ссылке |
| 5, 7 | Индикатор содержит либо OCI_IND_NULL, либо OCI_IND_NOTNULL; это специальные значения, определяемые во включаемом файле Oracle. В программном коде возвращаемому индикатору явно присваивается одно или другое значение |
| 11–12 | Возвращаемый индикатор обладает более высоким приоритетом, чем возврат 0; последний код просто игнорируется |

Простая команда для компиляции и компоновки предыдущей программы на моей 64-разрядной машине с Oracle (с использованием GCC):

```
gcc -m64 extprocsh.c -fPIC -G -I$ORACLE_HOME/rdbms/public -o extprocsh.so
```

А вот как выглядит эквивалентная команда на машине с Microsoft Windows (должны вводиться в одной строке):

```
c:\MinGW\bin\gcc -Ic:\oracle\product\10.2.0\db_1\oci\include extprocsh.c
-shared -o extprocsh.dll
```

А теперь приготовьтесь — мы переходим к подробному изучению отображения параметров.

Отображение параметров: полная картина

Как было показано в предыдущем разделе, при перемещении данных между PL/SQL и С каждый тип данных PL/SQL соответствует внешнему типу данных, идентифицируемому ключевым словом PL/SQL, который в свою очередь отображается на набор разрешенных типов С.

Таблица 28.2. Допустимые отображения типов данных PL/SQL и С

| Тип данных параметра PL/SQL | Ключевое слово PL/SQL, идентифицирующее внешний тип | Типы данных С для параметров PL/SQL: N или возвращаемые значения функций | IN OUT, OUT или любые параметры, передаваемые по ссылке (BY REFERENCE) |
|--|--|--|--|
| Длинные целые: BINARY_INTEGER, BOOLEAN, PLS_INTEGER | INT, UNSIGNED INT, CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T | int, unsigned int, char, unsigned char, short, unsigned short, long, unsigned long, sb1, ub1, sb2, ub2, sb4, ub4, size_t | Совпадает со списком слева, но с указателем (например, int * вместо int) |
| Короткие целые: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE | То же, что выше, но по умолчанию используется UNSIGNED INT | То же, что выше, но по умолчанию используется unsigned int | То же, что выше, но по умолчанию используется unsigned int * |
| Символьные: VARCHAR2, CHAR, NCHAR, LONG, NVARCHAR2, VARCHAR, CHARACTER, ROWID | STRING , OCISTRING | char * , OCIStrng * | char * , OCIStrng * |
| NUMBER | OCINUMBER | OCINumber * | OCINumber * |
| DOUBLE PRECISION | DOUBLE | double | double * |
| FLOAT, REAL | FLOAT | float | float * |
| RAW, LONG RAW | RAW , OCIRAW | unsigned char * , OCIRaw * | unsigned char * , OCIRaw * |
| DATE | OCIDATE | OCIDate * | OCIDate * |
| Временные метки: TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE | OCIDATETIME | OCIDateTime * | OCIDateTime * |
| INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH | OCIINTERVAL | OCIInterval * | OCIInterval * |
| BFILE, BLOB, CLOB, NCLOB | OCILOBLOCATOR | OCILOBLOCATOR * | OCILOBLOCATOR ** |
| Дескриптор пользовательского типа (коллекции или объекта) | TDO | OCIType * | OCIType * |
| Значение пользовательской коллекции | OCICOLL | OCIColl **, OCIArray **, OCITable ** | OCIColl **, OCIArray **, OCITable ** |
| Значение пользовательского объекта | DVOID | dvoid * | dvoid * (используйте dvoid ** для нефинальных типов, передаваемых в режиме IN OUT или OUT) |

Внешний тип данных идентифицируется в секции **PARAMETERS** ключевым словом, известным для PL/SQL. Иногда имена внешних типов совпадают с именами типов C, в других случаях такого соответствия нет. Например, при передаче переменной PL/SQL типа **PLS_INTEGER** внешним типом по умолчанию будет тип **INT**, соответствующий **int** в C. Однако тип **VARCHAR2** в Oracle использует внешний тип данных **STRING**, которому обычно соответствует тип **char *** в C.

В табл. 28.2 перечислены все преобразования типов данных, поддерживаемые интерфейсом PL/SQL–C в Oracle. Разрешенные преобразования зависят как от типа данных, так и от режима передачи формального параметра PL/SQL, как показано в предыдущем примере. Значения по умолчанию выделены в таблице жирным шрифтом.

В некоторых простых ситуациях — например, если передаются только числовые аргументы, для которых доступны значения по умолчанию, — секцию **PARAMETERS** можно вообще опустить. Тем не менее она должна использоваться при передаче индикаторов или других свойств данных.

Каждый передаваемый блок информации передается в отдельном параметре и присутствует как в секции **PARAMETERS**, так и в спецификации функции языка C.

Секция **PARAMETERS**

Секция **PARAMETERS** содержит список, разделенный запятыми, который может содержать пять разных видов элементов:

- имя параметра, за которым следует идентификатор внешнего типа данных;
- ключевое слово **RETURN** и связанный с ним идентификатор внешнего типа данных;
- «свойство» параметра PL/SQL или возвращаемого значения — например, индикатор **NULL** или целочисленная длина;
- ключевое слово **CONTEXT**, обозначающее указатель на контекст;
- ключевое слово **SELF** для внешних процедур, являющихся методами объектных типов.

Синтаксис элементов (за исключением **CONTEXT**) строится по следующей схеме:

`{имя | RETURN | SELF} [свойство] [BY REFERENCE] [внешний_тип_данных]`

Если спецификация вызова включает секцию **WITH CONTEXT**, то соответствующий элемент в списке параметров выглядит просто:

CONTEXT

По правилам при наличии **WITH CONTEXT** аргумент **CONTEXT** должен стоять на первом месте — на случай, если остальные отображения параметров задаются по умолчанию.

Смысл обозначений параметров:

- *имя* | **RETURN** | **SELF** — имя параметра в том виде, в котором оно задано в списке формальных параметров модуля PL/SQL, или ключевое слово **RETURN**, или ключевое слово **SELF** (для методов объектных типов). Имена параметров PL/SQL не обязаны совпадать с именами формальных параметров функции языка C. Однако параметры в списке параметров PL/SQL должны однозначно соответствовать параметрам спецификации языка C.
- *свойство* — одна из следующих конструкций: **INDICATOR**, **INDICATOR STRUCT**, **LENGTH**, **MAXLEN**, **TDO**, **CHARSETID** или **CHARSETFORM**. Они описаны в следующем разделе.
- **BY REFERENCE** — параметр передается по ссылке. Иначе говоря, модуль общей библиотеки ожидает получить указатель на параметр вместо его значения. Секция **BY REFERENCE** имеет смысл только для скалярных параметров **IN**, которые не являются строками, — например, **BINARY_INTEGER**, **PLS_INTEGER**, **FLOAT**, **DOUBLE PRECISION** и **REAL**. Все

остальные параметры (IN OUT и OUT, а также параметры IN типа STRING) всегда передаются по ссылке, а в соответствующем прототипе C должен использоваться указатель.

- *внешний тип данных* — ключевое слово внешнего типа данных из второго столбца табл. 28.2. Если оно не указано, то внешний тип данных выбирается по умолчанию в соответствии в таблице.

Свойства PARAMETERS

В этом разделе описаны все возможные свойства, которые могут определяться в секции PARAMETERS.

Свойство INDICATOR

Свойство INDICATOR — флаг, обозначающий равенство параметра NULL. Обладает следующими характеристиками:

- Допустимые внешние типы — short (по умолчанию), int и long.
- Допустимые типы PL/SQL — все скалярные типы могут использовать INDICATOR; для передачи индикаторных переменных для составных типов (таких, как пользовательские объекты и коллекции) используется свойство INDICATOR STRUCT.
- Допустимые режимы PL/SQL — IN, IN OUT, OUT и RETURN.
- Режим передачи — по значению для параметров IN (если только не задано условие BY REFERENCE), по ссылке для переменных IN OUT, OUT и RETURN.

Свойство может применяться к любому параметру в любом режиме, включая RETURN. Если индикатор отсутствует, PL/SQL вроде бы должен считать, что значение во внешней функции всегда отлично от NULL (но на самом деле все не так просто; см. далее врезку «Индикация без индикаторов?»).

ИНДИКАЦИЯ БЕЗ ИНДИКАТОРОВ?

Что произойдет, если не указать индикаторную переменную для строки, а затем вернуть пустую строку C? Чтобы получить ответ, я написал короткую тестовую программу:

```
void mynull(char *outbuff){
    outbuff[0] = '\0';
}
```

Спецификация вызова может выглядеть так:

```
CREATE OR REPLACE PROCEDURE mynull
    (res OUT VARCHAR2)
AS
    LANGUAGE C
    LIBRARY mynulllib
    NAME "mynull";
```

При вызове внешней процедуры PL/SQL не интерпретирует значение этого параметра как NULL. Похоже, внешний тип STRING интерпретируется особым образом; вы также можете обозначить значение NULL для Oracle, передав строку длины 2, в которой первый байт равен \0. (Этот способ работает только при опущенном параметре LENGTH.)

Но я не рекомендую использовать это решение «для ленивых»; лучше используйте индикатор!

При передаче внешней процедуре переменной IN и связанного с ней индикатора Oracle задает значение индикатора автоматически. Но если модуль C возвращает

значение в параметре RETURN или OUT и индикатор, ваш код C должен задать значение индикатора.

Для параметра IN параметр-индикатор в функции C может выглядеть так:

```
short pIndicatorFoo
```

Для параметра IN OUT:

```
short *pIndicatorFoo
```

В теле функции C следует использовать константы OCI_IND_NOTNULL и OCI_IND_NULL, которые становятся доступными для программы при включении oci.h. Oracle определяет их следующим образом:

```
typedef sb2 OCIInd;
#define OCI_IND_NOTNULL (OCIInd)0          /* not NULL */
#define OCI_IND_NULL (OCIInd)(-1)         /* NULL */
```

Свойство LENGTH

Свойство LENGTH содержит целое число с количеством символов в символьном параметре и обладает следующими характеристиками:

- Допустимые внешние типы — int (по умолчанию), short, unsigned short, unsigned int, long, unsigned long.
- Допустимые типы PL/SQL — VARCHAR2, CHAR, RAW, LONG RAW.
- Допустимые режимы PL/SQL — IN, IN OUT, OUT, RETURN.
- Режим передачи — по значению для параметров IN (если только не задано условие BY REFERENCE), по ссылке для переменных IN OUT, OUT и RETURN.

Свойство LENGTH является обязательным для RAW и LONG RAW и доступно в программах C для других типов данных в символьном семействе. При передаче RAW из PL/SQL в C Oracle задает свойство LENGTH; однако если вам потребуется вернуть данные RAW, свойство LENGTH должно задаваться программой C.

Для параметра IN параметр-индикатор в функции C может выглядеть так:

```
int pLenFoo
```

Для параметров OUT или IN OUT:

```
int *pLenFoo
```

Свойство MAXLEN

Свойство MAXLEN содержит целое число, обозначающее максимальное количество символов в строковом параметре, и обладает следующими характеристиками:

- Допустимые внешние типы — int (по умолчанию), short, unsigned short, unsigned int, long, unsigned long.
- Допустимые типы PL/SQL — VARCHAR2, CHAR, RAW, LONG RAW.
- Допустимые режимы PL/SQL — IN, IN OUT, OUT, RETURN.
- Режим передачи — по ссылке.

Свойство MAXLEN применяется только для параметров IN OUT, OUT и RETURN. При попытке использовать его для IN компилятор выдает ошибку PLS-00250 (недопустимое использование MAXLEN в секции параметров).

В отличие от параметра LENGTH, данные MAXLEN всегда передаются по ссылке.

Пример формального параметра C:

```
int *pMaxLenFoo
```


Свойства CHARSETID и CHARSETFORM

Свойства CHARSETID и CHARSETFORM представляют собой флаги с информацией о наборе символов. Они обладают следующими характеристиками:

- Допустимые внешние типы — unsigned int (по умолчанию), unsigned short, unsigned long.
- Допустимые типы PL/SQL — VARCHAR2, CHAR, CLOB.
- Допустимые режимы PL/SQL — IN, IN OUT, OUT, RETURN.
- Режим передачи — по ссылке.

При передаче внешней процедуре данных с набором символов, отличным от используемого по умолчанию, эти свойства позволяют передать вызываемой программе С идентификатор и форму набора символов. Значения доступны только для чтения; вызываемая программа не должна изменять их. Пример секции PARAMETERS с информацией о наборе символов:

```
PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR, cmd CHARSETID,
             cmd CHARSETFORM);
```

Oracle задает эти дополнительные значения автоматически на основании набора символов, в котором выражается аргумент cmd. За дополнительной информацией о поддержке глобализации Oracle в программах С обращайтесь к документации OCI.

Инициирование исключений из вызываемой программы С

Программа shell, приведенная ранее в этой главе, написана «в стиле С»: функция возвращает код состояния, по которому вызывающая сторона проверяет успешность вызова. А не лучше было бы реализовать программу (по крайней мере в PL/SQL) в виде процедуры, которая просто иницирует исключение при возникновении проблем? Давайте посмотрим, как реализуется OCI-эквивалент RAISE_APPLICATION_ERROR.

Кроме простого преобразования функции в процедуру, необходимо решить ряд задач:

- Передать контекстную область.
- Выбрать сообщение об ошибке и код ошибки в диапазоне 20001–20999.
- Добавить вызов служебной функции OCI, которая иницирует исключение.

Изменения в спецификации вызова тривиальны:

```
/* Файл в Сети: extprocsh.sql */ PROCEDURE shell(cmd IN VARCHAR2)
AS
  LANGUAGE C
  LIBRARY extprocshell_lib
  NAME "extprocsh"
  WITH CONTEXT
  PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR);
/
```

(Я также удалил возвращаемый параметр и его индикатор.) Следующий код показывает, как получать и использовать указатель на контекст в вызове, необходимом для инициирования исключения:

```
/* Файл в Сети: extprocsh.c */
1  #include <ociextp.h>
2  #include <errno.h>
3
4  void extprocsh(OCIExtProcContext *ctx, char *cmd, short cmdInd)
5  {
6      int excNum = 20001; # a convenient number :->
7      char excMsg[512];
```

```

8      size_t excMsgLen;
9
10     if (cmdInd == OCI_IND_NULL)
11         return;
12
13     if (system(cmd) != 0)
14     {
15         sprintf(excMsg, "Error %i during system call: %.s", errno, 475,
16                 strerror(errno));
17         excMsgLen = (size_t)strlen(excMsg);
18
19         if (OCIExtProcRaiseExcpWithMsg(ctx, excNum, (text *)excMsg, excMsgLen)
20             != OCIEXTPROC_SUCCESS)
21             return;
22     }
23
24 }
```

Ключевые аспекты этого кода перечислены в следующей таблице.

| Строки | Описание |
|--------|--|
| 4 | Первый формальный параметр содержит указатель на контекст. |
| 6 | Можно использовать любое число из диапазона пользовательских ошибок Oracle; обычно я не рекомендую жестко фиксировать эти значения в программе, но это всего лишь учебный пример |
| 7 | Максимальный размер текста пользовательских исключений равен 512 байтам |
| 8 | Переменная для хранения длины текста сообщения об ошибке; будет использоваться в вызове OCI, инициирующем исключение |
| 10–11 | Семантика NULL-аргумента приведенной ранее функции преобразуется в семантику процедуры: при вызове с параметром NULL не происходит ничего |
| 13 | Если возвращается ноль, значит, выполнение прошло успешно; ненулевой код соответствует ошибке или предупреждению |
| 15, 17 | Подготовка переменных с сообщением об ошибке и его длиной |
| 19–20 | Указатель на контекст используется функцией OCI, непосредственно инициирующей исключение |

И как это все откомпилировать? Для начала приведу команду для Unix/Linux:

```
gcc -m64 -extprocsh.c -I$ORACLE_HOME/rdbms/public -fPIC -shared -o extprocsh.so
```

Достаточно просто. Но как оказалось, в Microsoft Windows для этого потребуется создать файл .def для определения точки входа (а вернее, для исключения потенциальных точек входа, входящих в библиотеку Oracle oci.lib):

```

/* Файл в Сети: build_extprocsh.bat */
echo LIBRARY extprocsh.dll > extprocsh.def
echo EXPORTS >> extprocsh.def
echo extprocsh >> extprocsh.def
```

Следующая команда разбита на несколько строк по ширине страницы, но вообще она должна вводиться в одной длинной строке:

```

c:\MinGW\bin\gcc -c extprocsh.def extprocsh.c -IC:\oracle\product\10.2.0\
db_1\oci\
include C:\oracle\product\10.2.0\db_1\oci\lib\msvc\oci.lib-shared -o
extprocsh.dll
```

При тестировании функции *должен* выдаваться следующий результат:

```

SQL> CALL shell('garbage');
CALL shell('garbage')
*
```

```
ERROR at line 1:
```

```
ORA-20001: Error 2 during system call: No such file or directory
```

То есть вы должны получить пользовательское исключение -20001 с соответствующим текстом «no such file or directory». К сожалению, я обнаружил, что `system` не всегда возвращает осмысленные коды ошибок, и на некоторых платформах выдается сообщение *ORA-20001: Error 0 during system call: Error 0*. (Вероятно, проблема решается использованием другого вызова вместо `system` — если хотите, займитесь.)

Некоторые функции OCI используются только при написании внешних процедур:

- `OCIExtProcAllocCallMemory` — выделяет память, которую Oracle автоматически освободит при возврате управления PL/SQL.
- `OCIExtProcRaiseExcp` — инициирует предопределенное исключение по номеру ошибки Oracle.
- `OCIExtProcRaiseExcpWithMsg` — инициирует пользовательское исключение с пользовательским сообщением об ошибке (см. предыдущий пример).
- `OCIExtProcGetEnv` — разрешает внешней процедуре выполнять обратные вызовы OCI к базе данных для выполнения кода SQL или PL/SQL.

Все этим функциям необходим указатель на контекст. За подробной документацией и примерами использования этих функций обращайтесь к руководству *Oracle Application Developer's Guide — Fundamentals*.

Нестандартные агенты

В Oracle9i Database появилась возможность запуска агентов внешних процедур через каналы, связанные с другими локальными серверами баз данных. Эта функциональность позволяет распределять нагрузку, связанную с выполнением высокочастотных внешних программ, на другие экземпляры баз данных.

Даже без других серверов выполнение внешней процедуры через нестандартного (nondefault) агента приводит к запуску отдельного процесса. Это может быть удобно для ненадежных внешних программ: запуск через нестандартного агента означает, что даже в случае сбоя его процесса `extproc` это никак не повлияет на другие внешние процедуры, выполняемые в сеансе.

Простой пример нестандартного агента: следующая конфигурация позволяет агенту выполняться в той же базе данных, но в отдельной задаче `extproc`. В файл `tnsnames.ora` необходимо включить дополнительную запись:

```
agent0 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = IPC)(KEY=ключ))
    (CONNECT_DATA = (SID = PLSExtProc))
  )
```

Здесь *ключ* может совпадать с используемым в записи `EXTPROC_CONNECTION_DATA`.

Так как агенты создаются со связью с базой данных, мы должны будем создать одну из них:

```
SQL> CREATE DATABASE LINK agent0link
2 CONNECT TO username IDENTIFIED BY password
3 USING 'agent0';
```

Наконец, агент может присутствовать в команде `CREATE LIBRARY`:

```
CREATE OR REPLACE LIBRARY extprocshell_lib_with_agent
AS 'c:\oracle\admin\local\lib\extprocsh.dll'
AGENT 'agent0';
```

Любая спецификация вызова, написанная для использования этой библиотеки, выполнит аутентификацию и подключение через связь `agent0`, с запуском задачи `extproc` отдельно от задачи `extproc` по умолчанию. Таким образом задачи отделяются друг от

друга (например, потоково-безопасные задачи могут передаваться многопоточным агентам, а все остальные — специализированным).

Oracle также поддерживает динамический механизм, позволяющий передать имя агента в параметре внешней процедуры. Чтобы использовать эту возможность, включите секцию **AGENT IN** в спецификацию вызова, например (изменения выделены жирным шрифтом):

```
CREATE OR REPLACE PROCEDURE shell2 (name_of_agent IN VARCHAR2, cmd VARCHAR2)
AS
  LANGUAGE C
  LIBRARY extprocshell_lib
  NAME "extprocsh2"
  AGENT IN (name_of_agent)
  WITH CONTEXT
  PARAMETERS (CONTEXT, name_of_agent STRING, cmd STRING, cmd INDICATOR);
```

Обратите внимание: имя агента включено в список параметров. Oracle требует, чтобы каждый формальный параметр имел соответствующий элемент в секции **PARAMETERS**, поэтому мне придется изменять внешнюю библиотеку C. В моем случае я просто добавляю вторую точку входа `extprocsh2` в библиотеку при помощи следующей тривиальной функции:

```
void extprocsh2(OCIExtProcContext *ctx, char *agent, char *cmd, short cmdInd)
{
    extprocsh(ctx, cmd, cmdInd);
}
```

В моем коде строка агента просто игнорируется. Теперь процедура `shell2` вызывается следующим образом:

```
CALL shell2('agent0', 'что угодно');
```

Если вы хотите, чтобы хранящая программа каким-то образом вызывала внешнюю процедуру на удаленной машине, у вас имеется пара вариантов. Либо реализуйте внешнюю процедуру на локальной машине и сделайте ее «сквозной» программой, которая будет выполнять удаленный вызов процедуры на базе C, либо реализуйте хранимую программу PL/SQL на удаленной машине в виде внешней процедуры и вызывайте ее с локальной машины через связь с базой данных. Во всяком случае, настроить слушателя внешней процедуры для приема сетевых подключений с другой машины вам не удастся.

ОДИССЕЯ ОТЛАДКИ

Некоторые отладчики, в числе которых отладчик GNU (GDB), могут подключаться к работающему процессу для отладки внешних процедур. Ниже приводится краткое описание того, как я делал это в Solaris и Windows XP.

На обеих платформах я начал с предварительной компиляции файла общей библиотеки с ключом компилятора (`-g` в случае GCC), необходимым для включения информации символьческих имен. Это была самая простая часть. В процессе тестирования «неожиданно» выяснилось, что 64-разрядную внешнюю процедуру Solaris нельзя отлаживать в 32-разрядном отладчике, так что мне пришлось построить и установить 64-разрядную исполняемую версию gdb. Эта фаза началась с выполнения в корневом каталоге дерева GDB команды:

```
OS> CC="gcc -m64" ./configure
```

Построение GDB преподнесло много сюрпризов. Наверное, все было бы намного проще, если бы у меня за спиной стоял компетентный системный администратор!

Другой подготовительный шаг (хотя и не обязательный) связан с запуском сценария

dbgextp.sql, который должен находиться в каталоге plsql/demo. В Oracle Database 10g и выше вы не найдете этот каталог в стандартном каталоге \$ORACLE_HOME, потому что весь каталог был перемещен на диск Companion CD. Впрочем, вам, возможно, удастся извлечь каталог plsql/demo следующей командой (снова разбитой по ширине страницы для удобства чтения):

```
OS> jar xvf /cdrom/stage/Components/oracle.rdbms.companion/10. x.x.x.x  
/1/DataFiles/filegroup1.jar
```

Файл dbgextp.sql содержит полезные комментарии, которые определенно стоит почитать. Запустите этот сценарий от своего имени (не от имени SYS) для построения пакета DEBUG_EXTPROC. Этот пакет содержит процедуру, единственным предназначением которой является запуск агента внешней процедуры; это позволяет узнать соответствующий идентификатор процесса (PID). В сеансе SQL*Plus она запускается следующим образом:

```
SQL> EXEC DEBUG_EXTPROC.startup_extproc_agent
```

Команда запускает процесс extproc, после чего вы можете узнать его PID командой ps -ef или pgrep extproc.

Почему я говорю, что пакет DEBUG_EXTPROC не является обязательным? Потому что агента также можно запустить, выполняя любую старую внешнюю процедуру, или, если вы используете многопоточных агентов, процесс будет создан заранее, и вы сможете узнать PID с минимальными усилиями.

Как бы то ни было, зная PID, вы можете запустить отладчик и присоединить его к выполняемому процессу:

```
OS> gdb $ORACLE_HOME/bin/extproc pid
```

При первой попытке я получил ошибку «отказано в доступе». Проблема была решена входом с учетной записи oracle.

Затем я установил точку прерывания на символическом имени rextproc согласно инструкциям из файла dbgextp.sql. Далее в сеансе SQL*Plus внешняя процедура была выполнена командой:

```
SQL> CALL shell(NULL);
```

Я отдал команду на продолжение работы GDB, и в ходе выполнения extproc была быстро достигнута точка прерывания rextproc. Затем я выполнил команду GDB share, чтобы отладчик прочитал символические имена из загруженной внешней библиотеки; наконец, я смог установить точку прерывания во внешней процедуре extprocdsh, выдал команду продолжения — и я нахожусь в своем коде! После этого все работало достаточно хорошо: я мог в пошаговом режиме выполнять свой код, просматривать переменные и т. д.

Я обнаружил, что отладка внешней процедуры в Cygwin GB на платформах Microsoft требовала внесения следующих изменений:

- Мне пришлось модифицировать службу слушателя из панели управления Windows, чтобы она выполнялась с правами моей учетной записи (вместо прав «Локальной системы»).
- Вместо команды ps -ef для получения PID extproc я воспользоваться программой Microsoft tasklist.exe.
- Чтобы просмотреть исходный код внешней процедуры во время сеанса отладки, я должен был запустить GDB из каталога, содержащего исходный файл (вероятно, это можно сделать и другим способом).
- В системе Solaris тестирование проводилось с 64-разрядной сборкой GDB 6.3 в Solaris 2.8. На компьютере с Windows XP я использовал двоичный файл Cygwin GDB 6.3 без всяких проблем, но заставить работать двоичный файл MinGW GDB 5.2.1 мне так и не удалось.

Сопровождение внешних процедур

Напоследок приведу несколько замечаний по поводу создания, отладки и сопровождения внешних процедур.

Удаление библиотек

Синтаксис удаления библиотеки очень прост:

```
DROP LIBRARY имя_библиотеки;
```

Пользователь Oracle, выполняющий эту команду, должен обладать привилегиями `DROP LIBRARY` или `DROP ANY LIBRARY`.

Oracle не проверяет информацию зависимостей перед удалением библиотеки. Этот факт очень полезен, если вам понадобится изменить имя или местоположение файла общего объекта, на который ссылается библиотека. Просто удалите и постройте его заново, и все зависимые функции продолжают нормально работать. (Наверное, еще полезнее было бы потребовать использования команды `DROP LIBRARY FORCE`, но такой возможности не существует.)

Прежде чем удалять библиотеку, возможно, стоит заглянуть в представление `DBA_DEPENDENCIES` и проверить, зависят ли от нее какие-либо модули PL/SQL.

Словарь данных

Словарь данных содержит несколько представлений для управления внешними процедурами. В табл. 28.3 приведены версии `USER_` таблиц словаря — но обратите внимание на соответствующие элементы для `DBA_` и `ALL_`.

Таблица 28.3. Представления словаря данных для внешних процедур

| Для получения ответа на вопрос... | Используется это представление | Пример |
|---|--------------------------------|--|
| Какие библиотеки я создал? | <code>USER_LIBRARIES</code> | <pre>SELECT * FROM user_libraries;</pre> |
| Какие хранимые программы PL/SQL используют библиотеку хуз в спецификации вызова? | <code>USER_DEPENDENCIES</code> | <pre>SELECT * FROM user_dependencies WHERE referenced_name = 'XYZ';</pre> |
| Какие агенты внешних процедур (специализированные и многопоточные) выполняются в настоящее время? | <code>V\$HS_AGENT</code> | <pre>SELECT * FROM V\$HS_AGENT WHERE UPPER(program) LIKE 'EXTPROCS%'</pre> |
| Какие сеансы Oracle используют тех или иных агентов? | <code>V\$HS_SESSION</code> | <pre>SELECT s.username, h.agent_id FROM V\$SESSION s, V\$HS_SESSION h WHERE s.sid = h.sid;</pre> |

Правила и предупреждения

Как это обычно бывает в PL/SQL, с использованием внешних процедур связан целый список предупреждений:

- Хотя с режимом каждого формального параметра (`IN`, `IN OUT`, `OUT`) в PL/SQL могут быть связаны определенные ограничения, С игнорирует эти режимы. Различия между режимом параметров PL/SQL и их использованием в модуле С не могут быть обнаружены на стадии компиляции и могут остаться незамеченными во вре-

мя выполнения. Правила выглядят так, как и следовало ожидать: не присваивайте значения параметрам **IN**, не читайте значения параметров **OUT**, всегда присваивайте значения параметрам **IN OUT** и **OUT**, всегда возвращайте значение соответствующего типа данных.

- Изменяемые значения **INDICATOR** и **LENGTH** всегда передаются по ссылке для **IN OUT**, **OUT** и **RETURN**. Неизменяемые значения **INDICATOR** и **LENGTH** всегда передаются по значению, если только не задана конструкция **BY REFERENCE**. Но даже если вы передаете значения **INDICATOR** или **LENGTH** для переменных **PL/SQL** по ссылке, они все равно остаются параметрами только для чтения.
- Теоретически между **PL/SQL** и **C** можно передавать до 128 параметров, но если какие-либо из них относятся к типу **float** или **double**, реальный максимум будет ниже. Насколько ниже — зависит от вашей операционной системы.
- Если вы используете многопоточных агентов (поддерживаемых в Oracle Database 10g и выше), у программы появляются дополнительные ограничения. Все вызовы из программы **C** должны быть безопасными по отношению к потокам. Кроме того, лучше избегать использования глобальных переменных **C**. Даже в однопоточной версии глобальные переменные могут работать не так, как ожидалось, из-за механизма «кэширования **DLL**» операционной системы.
- Внешняя процедура не может выполнять команды **DDL**, начинать или завершать сеансы, управлять транзакциями с использованием **COMMIT** или **ROLLBACK**. (За списком неподдерживаемых функций **OCI** обращайтесь к документации *Oracle PL/SQL User's Guide and Reference*.)



Параметры функций и метасимволы регулярных выражений

В этом приложении описаны различные метасимволы регулярных выражений, поддерживаемые в Oracle Database 10g и выше. Также приводится сводка синтаксиса функций REGEXP_. За дополнительной информацией о поддержке регулярных выражений в Oracle обращайтесь к главе 8.

Метасимволы

В столбце «Версия» в табл. А.1–А.3 указано, в какой версии появился тот или иной метасимвол: Oracle Database 10g Release 1 или Release 2.

Таблица А.1. Метасимволы совпадений

| Синтаксис | Версия | Описание |
|-------------|--------|--|
| . | 10gR1 | Совпадает с любым одиночным символом, кроме новой строки. Совпадает с символом новой строки при установленном флаге n. На платформах Windows, Linux и Unix chr(10) интерпретируется как символ новой строки |
| [...] | 10gR1 | Определяет список, который совпадает с любым символом, перечисленным в квадратных скобках. Допускается определение диапазонов символов (например, a-z). Диапазоны интерпретируются в зависимости от параметра NLS_SORT. Дефис (-) интерпретируется как литерал, если он находится на первом или последнем месте в списке (например, [abc-]). Закрывающая квадратная скобка (]) интерпретируется как литерал, если она находится на первом месте в списке (например, []abc]). Если список начинается с каретки (^), то он инвертируется (см. следующий элемент) |
| [^...] | 10gR1 | Совпадает с любым символом, не перечисленным в скобках (называется <i>списком несовпадений</i>) |
| [[:класс:]] | 10gR1 | Совпадает с любым символом, входящим в заданный символьный класс. Может использоваться только в списках совпадений: [[[:класс:]]abc] является допустимым выражением, а [:class:]abc — нет. Допустимые имена символьных классов перечислены в табл. А.5 |
| [.комб.] | 10gR1 | Совпадает с заданным комбинированным элементом, который может состоять из одного или нескольких символов. Может использоваться только в списках совпадений. Например, выражение [[.ch.]] совпадает с испанской буквой ch. Комбинированные элементы перечислены в табл. А.4 |
| [=симв=] | 10gR1 | Совпадает с любыми символами, использующими заданный символ как базовый. Может использоваться только в списках совпадений. Например, [[=e=]] совпадает с любым из символов: «eéëèÊË» |
| \d | 10gR2 | Совпадает с любой цифрой, эквивалент [[:digit:]] |

| Синтаксис | Версия | Описание |
|-----------|--------|--|
| \D | 10gR2 | Совпадает с любым нецифровым символом, эквивалент <code>[^[:digit:]]</code> |
| \w | 10gR2 | Совпадает с любым «символом слова». К этой категории относятся алфавитные символы, цифровые символы и символ подчеркивания |
| \W | 10gR2 | Совпадает с любым символом, не являющимся «символом слова» |
| \s | 10gR2 | Совпадает с любым символом пропуска, эквивалент <code>[[:space:]]</code> |
| \S | 10gR2 | Совпадает с любым символом, не относящимся к категории пропусков, эквивалент <code>[^[:space:]]</code> |

Таблица А.2. Квантификаторы

| Синтаксис | Версия | Описание |
|-----------|--------|---|
| ? | 10gR1 | Ноль или одно совпадение |
| * | 10gR1 | Ноль и более совпадений |
| + | 10gR1 | Одно и более совпадений |
| {m} | 10gR1 | Ровно m совпадений |
| {m,} | 10gR1 | Не менее m совпадений |
| {m,n} | 10gR1 | Не менее m, не более n совпадений |
| +? | 10gR2 | Одно и более совпадений в минимальном режиме |
| ?? | 10gR2 | Ноль или одно совпадение в минимальном режиме |
| {m}? | 10gR2 | То же, что {m} |
| {m,}? | 10gR2 | Не менее m совпадений в минимальном режиме (поиск прекращается при достижении m совпадений) |
| {m,n}? | 10gR2 | Не менее m, не более n совпадений в минимальном режиме; если возможно, находится ровно m совпадений |

Таблица А.3. Другие метасимволы

| Синтаксис | Версия | Описание |
|-----------|--------|---|
| | 10gR1 | Альтернатива. В подвыражениях альтернатива не выходит за пределы подвыражения |
| (...) | 10gR1 | Определяет подвыражение |
| \n | 10gR1 | Ссылается на текст, совпавший с n-м подвыражением. Обозначения обратных ссылок лежат в диапазоне от \1 до \9 |
| \ | 10gR1 | Если за символом \ не следует цифра, он считается экранирующим (escape) символом. Например, шаблон \1 обозначает одиночный литерал \, за которым следует цифра 1; для поиска открывающих скобок используется комбинация \((|
| ^ | 10gR1 | Привязка выражения к началу символьной последовательности (в многострочном режиме — к началу строки) |
| \$ | 10gR1 | Привязка выражения к концу символьной последовательности (в многострочном режиме — к концу строки) |
| \A | 10gR2 | Привязка выражения к началу строки независимо от действия многострочного режима |
| \Z | 10gR2 | Привязка выражения к концу строки или символу новой строки, который завершает строку (независимо от действия многострочного режима) |
| \z | 10gR2 | Привязка выражения к концу строки независимо от действия многострочного режима |

Таблица А.4. Комбинированные элементы

| NLS_SORT | Многосимвольные комбинированные элементы | | |
|-----------|--|----------|----------|
| XCROATIAN | d_ lj | D_ LJ | D_ Lj |

Таблица А.4 (продолжение)

| NLS_SORT | Многосимвольные комбинированные элементы | | |
|--------------------|--|----|----|
| | nj | Nj | NJ |
| XCZECH | Ch | CH | Ch |
| XCZECH_PUNCTUATION | Ch | CH | Ch |
| XDANISH | aa | AA | Aa |
| | oe | OE | Oe |
| XHUNGARIAN | cs | CS | Cs |
| | gy | GY | Gy |
| | ly | LY | Ly |
| | ny | NY | Ny |
| | sz | SZ | Sz |
| | ty | TY | Ty |
| | zs | ZS | Zs |
| XSLOVAK | dz | DZ | Dz |
| | d_ | D_ | D_ |
| | ch | CH | Ch |
| XSPANISH | ch | CH | Ch |
| | ll | LL | Ll |

Таблица А.5. Поддерживаемые символьные классы

| Класс | Описание |
|--------------|---|
| [[:alnum:]] | Алфавитно-цифровые символы (то же, что [[:alpha:]] + [[:digit:]]) |
| [[:alpha:]] | Только алфавитные символы |
| [[:blank:]] | Пустые символы (пробелы, табуляции и т. д.) |
| [[:cntrl:]] | Непечатаемые (управляющие) символы |
| [[:digit:]] | Обычные цифры |
| [[:graph:]] | Графические символы (то же, что [[:punct:]] + [[:upper:]] + [[:lower:]] + [[:digit:]]) |
| [[:lower:]] | Буквы нижнего регистра |
| [[:print:]] | Печатаемые символы |
| [[:punct:]] | Знаки препинания |
| [[:space:]] | Пропуски (пробел, прогон страницы, новая строка, возврат курсора, горизонтальная табуляция, вертикальная табуляция) |
| [[:upper:]] | Буквы верхнего регистра |
| [[:xdigit:]] | Шестнадцатеричные цифры |

Функции и параметры

В следующих разделах представлен синтаксис функций регулярных выражений Oracle. Смысл параметров описан в разделе «Параметры регулярных выражений» (см. далее).

Функции регулярных выражений

REGEXP_COUNT (Oracle Database 11g и выше)

Возвращает набор совпадений выражения в целевой строке. Синтаксис:

```
REGEXP_COUNT(целевая_строка, выражение
              [, позиция
              [, параметры_совпадения]])
```

REGEXP_INSTR

Возвращает позицию совпадения регулярного выражения в целевой строке. Синтаксис:

```
REGEXP_INSTR(целевая_строка, выражение  
             [, позиция [, вхождение  
             [, режим_возврата  
             [, параметры_совпадения  
             [, подвыражение]]]])
```

REGEXP_LIKE

Определяет, содержит ли заданная строка текст, совпадающий с выражением. Это логическая функция, возвращающая TRUE, FALSE или NULL. Синтаксис:

```
REGEXP_LIKE (целевая_строка, выражение  
            [, параметры_совпадения])
```

REGEXP_REPLACE

Выполняет операцию поиска и замены (за подробностями обращайтесь к главе 8). Синтаксис:

```
REGEXP_REPLACE(целевая_строка, выражение  
              [, строка_замены  
              [, позиция [, вхождение  
              [, параметры_совпадения]]]])
```

REGEXP_SUBSTR

Извлекает текст, совпадающий с регулярным выражением в строке. Синтаксис:

```
REGEXP_SUBSTR(целевая_строка, выражение  
             [, позиция [, вхождение  
             [, параметры_совпадения  
             [, подвыражение]]]])
```

Параметры регулярных выражений

Параметры, передаваемые функциям регулярных выражений, описаны ниже:

- *целевая_строка* — строка, в которой производится поиск совпадения.
- *выражение* — регулярное выражение, описывающее искомый текст.
- *строка_замены* — строка, генерирующая текст, который должен использоваться в операции поиска с заменой.
- *позиция* — позиция символа в целевой строке, с которой начинается поиск. По умолчанию равна 1.
- *вхождение* — номер вхождения искомого совпадения. По умолчанию равен 1 (поиск первого возможного совпадения).
- *режим_возврата* — используется только для REGEXP_INSTR; определяет, позиция какого символа должна возвращаться для совпадения. По умолчанию используется значение 0 (начало). Используйте значение 1, чтобы возвращать конечную позицию.
- *параметры_совпадения* — текстовая строка с параметрами, управляющими процессом поиска совпадений:
 - *c* — поиск с учетом регистра символов (по умолчанию определяется значением NLS_SORT).
 - *i* — поиск без учета регистра символов.
 - *n* — разрешает совпадение точки с символами новой строки. По умолчанию точка не совпадает с новой строкой.

- `m` — изменяет определение строки в контексте метасимволов `^` и `$`. По умолчанию под строкой понимается вся целевая строка. С параметром `m` под строкой понимается каждая из логических строк, ограниченных символами новой строки.
- *подвыражение* (Oracle Database 11g и выше) — цифра (0–9), идентифицирующая подвыражение. По умолчанию используется значение 0, означающее, что подвыражения не используются.

Вы можете указать несколько параметров совпадения в произвольном порядке. Например, строка `<ip>` означает то же, что `<pi>`. Если задать конфликтующие параметры (например, `<is>`), будет использован последний параметр (`<с>` в данном случае).

Б

Числовые форматы

Числовые форматы используются с функциями `TO_CHAR` и `TO_NUMBER`. При вызове `TO_CHAR` они указывают, как именно числовое значение должно преобразовываться в строку `VARCHAR2`. Вы можете указать разделитель, местоположение знака для положительного или отрицательного числа, а также другие полезные атрибуты. И наоборот, в вызовах `TO_NUMBER` числовые форматы определяют способ интерпретации числовых значений.

Маска числового формата содержит один или несколько элементов из табл. Б.1. Полученная строка (или преобразованное числовое значение) представляет комбинацию используемых элементов форматной маски. Примеры разных применений форматных масок представлены в описаниях функций `TO_CHAR` и `TO_NUMBER`.

Если описание форматного элемента начинается с «Префикс:», то этот элемент может использоваться только в начале форматной маски; если описание начинается с «Суффикс:», значит, элемент может использоваться только в конце форматной маски. Большинство форматных элементов описывается в контексте их влияния на преобразование символа в строковое представление. Учтите, что большинство этих элементов также может использоваться в обратном направлении — для определения формата символьной строки, преобразуемой в число.

Таблица Б.1. Элементы маски числового формата

| Форматный элемент | Описание |
|-------------------|--|
| \$ | Префикс: перед числом выводится знак доллара (за информацией об условном знаке денежной единицы обращайтесь к описанию элемента C) |
| , (запятая) | Включает запятую в возвращаемое значение. Запятая используется для разделения групп разрядов (см. описание форматного элемента G) |
| . (точка) | Включает точку в возвращаемое значение. Точка используется как разделитель дробной части (см. описание форматного элемента D) |
| 0 | Каждый ноль представляет возвращаемую значащую цифру. Начальные нули в числе включаются в выходную строку |
| 9 | Каждая девятка представляет возвращаемую значащую цифру. Вместо начальных нулей в числе выводятся пробелы |
| B | Префикс: возвращает нулевое значение в виде пробелов, даже если в маске форматный элемент 0 используется для обозначения начальных нулей |
| C | Задаёт местонахождение знака денежной единицы ISO в возвращаемом значении. Параметр <code>NLS_ISO_CURRENCY</code> задаёт денежный знак ISO |
| D | Задаёт местонахождение разделителя дробной части в возвращаемом значении. Все форматные элементы справа от D используются для форматирования дробной части значения. Символ, используемый для разделителя дробной части, определяется параметром <code>NLS_NUMERIC_CHARACTERS</code> базы данных |

Таблица Б.1 (продолжение)

| Формат-ный элемент | Описание |
|--------------------|--|
| EEEE | Суффикс: означает, что число должно возвращаться в научной записи |
| FM | Префикс: удаляет начальные или завершающие пробелы из возвращаемого значения |
| G | Задаёт местонахождение разделителя групп разрядов (например, точка или запятая в 6,754 или 6.754) в возвращаемом значении. Символ, используемый для разделителя групп, определяется параметром NLS_NUMERIC_CHARACTERS базы данных |
| L | Задаёт местонахождение местного символа денежной единицы (например, \$ или €) в возвращаемом значении. Местный символ денежной единицы задается параметром NLS_CURRENCY |
| MI | Суффикс: если число отрицательно, после него выводится знак «минус» (-). Если число положительно, то после числа выводится завершающий пробел |
| PR | Суффикс: отрицательное значение заключается в угловые скобки (< >). Положительные значения выводятся с начальным и конечным пробелом |
| RN или rn | Возвращаемое значение преобразуется в римскую запись в верхнем или нижнем регистре. Преобразование в римскую запись возможно для чисел в диапазоне от 1 до 3999. Значение должно быть целым числом. RN возвращает римскую запись в верхнем регистре, а rn — римскую запись в нижнем регистре |
| S | Префикс: перед положительными значениями выводится знак «плюс» (+), перед отрицательными — знак «минус» (-) |
| TM | Префикс: число выводится с минимальным количеством символов (сокращение от «Text Minimum»). Если вам нужна обычная десятичная запись, поставьте за TM одну девятку. Для использования научной записи за TM должна следовать одна буква E |
| U | В заданной позиции выводится двойной денежный знак (часто €). Символ, возвращаемый для этого форматного элемента, определяется параметром NLS_DUAL_CURRENCY |
| V | Число слева от V в форматной маске умножается на 10 в n-й степени, где n — число знаков 9 справа от V в форматной модели |
| X | Возвращает число в шестнадцатеричной форме. Перед элементом можно поставить нули, чтобы обеспечить вывод начальных нулей, или FM для усечения начальных и конечных пробелов. X не может использоваться в сочетании с другими форматными элементами |

Следует учитывать, что иногда два элемента могут приводить к одинаковым результатам — на первый взгляд. Например, вы можете использовать знак доллара (\$), запятую (,) или точку (.) или же элементы L, G и D соответственно. Буквенные элементы соответствуют текущим настройкам NLS и возвращают символы для используемого языка. Например, в некоторых европейских языках дробная часть числа отделяется от целой запятой, а не точкой. Форматные элементы \$, , и . ориентированы на США и всегда возвращают эти три символа. Мы рекомендуем использовать NLS-совместимые элементы форматной маски (такие, как L, G и D), если только у вас нет веских причин поступать иначе.

Денежные единицы

В табл. Б.1 перечислены четыре форматных элемента, используемых для обозначения денежных единиц, — \$, L, S и U. Вероятно, вас интересует, чем они отличаются друг от друга?

- Форматный элемент \$ — всегда возвращает знак доллара (\$).
- Форматный элемент L — учитывает текущую настройку NLS_CURRENCY, в которую входит индикатор местной денежной единицы. Если, например, согласно настройке NLS_TERRITORY вы находитесь в Великобритании, в NLS_CURRENCY по умолчанию будет использоваться фунт (£), и при использовании форматного элемента L будет выводиться знак £.

- Форматный элемент `C` — аналогичен элементу `L`, но использует индикатор денежной единицы `ISO`, заданный текущей конфигурацией `NLS_ISO_CURRENCY`. Для Великобритании выводится сокращение `GBP`, для Соединенных Штатов — `USD` и т. д.
- Форматный элемент `U` — был добавлен для поддержки евро и использует индикатор денежной единицы, заданный текущей конфигурацией `NLS_DUAL_CURRENCY`. Для стран, поддерживающих евро, конфигурация `NLS_DUAL_CURRENCY` по умолчанию использует знак евро (€).

Для просмотра текущих настроек `NLS_CURRENCY` и `NLS_ISO_CURRENCY` следует обратиться с запросом к системным представлениям `NLS_SESSION_PARAMETERS` или `V$NLS_PARAMETERS`.

В Маска формата даты

В табл. В.1 перечислены элементы маски формата даты, используемой с функциями `TO_CHAR`, `TO_DATE`, `TO_TIMESTAMP` и `TO_TIMESTAMP_TZ`. Также можно задать форматы даты и временной метки по умолчанию на уровне сеанса — возможность, которая пригодится, если ваши конкретные потребности отличаются от потребностей большинства пользователей базы данных. Форматы даты и временной метки по умолчанию на уровне сеанса задаются командой `ALTER SESSION`. Следующий пример работает в Oracle8i Database и выше и задает формат даты по умолчанию `MM/DD/YYYY`:

```
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION SET NLS_DATE_FORMAT='''MM/DD/YYYY''';
END;
```

Чтобы проверить формат даты, действующий в вашем сеансе на данный момент, выдайте следующий запрос к представлению словаря данных `NLS_SESSION_PARAMETERS`:

```
SELECT value
FROM nls_session_parameters
WHERE parameter='NLS_DATE_FORMAT';
```

Чтобы назначить или проверить формат временной метки по умолчанию, используйте параметры `NLS_TIMESTAMP_FORMAT` и `NLS_TIMESTAMP_TZ_FORMAT`.

Некоторые элементы табл. В.1 применимы только при преобразовании значений даты/времени из внутреннего формата Oracle в символьные строки, но не наоборот. Такие элементы не могут использоваться в маске даты по умолчанию (например, с `NLS_DATE_FORMAT`), потому что она применяется к преобразованиям в обе стороны. Такие элементы в таблице снабжены пометкой «Только для вывода».

Таблица В.1. Элементы маски формата даты

| Элемент | Описание |
|---------------|--|
| Другой текст | Любые знаки препинания (запятая, косая черта, дефис) воспроизводятся в отформатированном результате преобразования. Чтобы произвольный текст воспроизводился при преобразовании точно в том виде, в каком он был введен, заключите его в двойные кавычки |
| A.M. или P.M. | Индикатор времени суток (утро или вечер) с точками |
| AM или PM | Индикатор времени суток (утро или вечер) без точек |
| B.C. или A.D. | Индикатор «до нашей эры/нашей эры» с точками |
| BC или AD | Индикатор «до нашей эры/нашей эры» без точек |
| CC и SCC | Век. При использовании формата SCC даты до нашей эры выводятся со знаком «-». Только для вывода |
| D | День недели, от 1 до 7. День недели, считающийся первым, неявно задается параметром инициализации <code>NLS_TERRITORY</code> в экземпляре базы данных |

| Элемент | Описание |
|------------------------|---|
| DAY, Day или day | Название дня недели в верхнем, смешанном или нижнем регистре |
| DD | День месяца, от 1 до 31 |
| DDD | День года, от 1 до 366 |
| DL | Длинный формат даты. Зависит от текущих значений NLS_TERRITORY и NLS_LANGUAGE. Может использоваться по отдельности или в сочетании с TS, но не с другими элементами |
| DS | Короткий формат даты. Зависит от текущих значений NLS_TERRITORY и NLS_LANGUAGE. Может использоваться по отдельности или в сочетании с TS, но не с другими элементами |
| DY, Dy или dy | Сокращенное название дня недели (например, TUE вместо Tuesday). |
| E | Сокращенное название эры. Допустимо только для календарей: японский имперский, официальный китайский, тайский буддистский |
| EE | Полное название эры |
| FF | Дробные секунды. Элемент действителен только со значениями TIMESTAMP. Количество возвращаемых цифр соответствует точности преобразуемой даты/времени. Всегда состоит из двух букв F независимо от количества цифр в дробной части, которые должны выводиться или использоваться. Любые другие количества недействительны |
| FF1..FF9 | То же, что FF, но цифра (1..9) определяет количество цифр в дробных секундах: FF1 для вывода одной цифры в дробной части, FF2 — для вывода двух цифр, и т. д. |
| FM | Управление подавлением пропусков при выводе (сокращение от «Fill Mode») |
| FX | Точное совпадение данных с маской формата (сокращение от «Format eXact») |
| HH или HH12 | Час дня, от 1 до 12. Только для вывода |
| HH24 | Час дня, от 0 до 23 |
| IW | Неделя года, от 1 до 52 или от 1 до 53 в зависимости от стандарта ISO. Только для вывода |
| IYY или IY или I | Последние три, две или одна цифра года в стандарте ISO. Только для вывода |
| IYYY | Год из четырех цифр в стандарте ISO. Только для вывода |
| J | Дата в формате юлианского календаря (количество дней с 1 января 4712 г. до н. э., самой ранней даты, поддерживаемой Oracle). |
| MI | Минуты в значении даты/времени, от 0 до 59. |
| MM | Номер месяца, от 01 до 12. Январь считается месяцем 01, сентябрь — 09, и т. д. |
| MON, Mon или mon | Сокращенное название месяца (например, JAN для января) в верхнем, смешанном или нижнем регистре |
| MONTH, Month или month | Название месяца в верхнем, смешанном или нижнем регистре |
| Q | Квартал года (от 1 до 4). Месяцы с января по март относятся к первому кварталу, с апреля по июнь — ко второму, и т. д. Только для вывода |
| RM | Номер месяца в римской записи, с I до XII: январь — I, сентябрь — IX, и т. д. |
| RR | Последние две цифры года. Используется для веков, отличных от текущего |
| RRRR | То же, что RR, при выводе; год из четырех цифр при вводе |
| SCC или CC | Век. С форматом SCC даты до нашей эры выводятся с префиксом «-». Только для вывода |
| SP | Суффикс преобразования числа в текстовый формат. Элемент может находиться в конце каждого элемента, возвращающего число. Например, для маски "DDth-Mon-Yyyysp" выводится результат "15th-Nov-One Thousand Nine Hundred Sixty-One". Возвращаемое значение всегда выводится на английском языке независимо от языка даты (обратите внимание: с элементом Yyyu слова выводятся в смешанном регистре) |
| SPTH или THSP | Суффикс преобразования числа в текстовый порядковый формат; например, 4 преобразуется в FOURTH, а 1 — в FIRST. Элемент может находиться в конце любого элемента, возвращающего число. Например, модель "Ddspth Mon, Yyyysp" выводит результат вида "Fifteenth Nov, One Thousand Nine Hundred Sixty-One". Возвращаемое значение всегда выводится на английском языке независимо от языка даты |

Таблица В.1 (продолжение)

| Элемент | Описание |
|--|---|
| SS | Второй компонент значения даты/времени, от 0 до 59 |
| SSSSS | Число секунд, прошедших с полуночи. Значения лежат в диапазоне от 0 до 86 399, каждый час состоит из 3600 секунд |
| SYEAR, YEAR, SYear, Year, syear или year | Год, записанный словами (например, «two thousand two»). С префиксом S перед датами до нашей эры выводится знак «-». Только для вывода |
| SYYYY или YYYY | Год из четырех цифр. Результат может выводиться в верхнем, смешанном или нижнем регистре |
| TH | Суффикс преобразования числа в порядковый формат; например, 4 преобразуется в 4th, а 1 — в 1st. Элемент может находиться в конце любого элемента, возвращающего число. Например, модель "DDth-Mon-YYYY" выводит результат вида "15th-Nov-1961". Возвращаемое значение всегда выводится на английском языке независимо от языка даты |
| TS | Короткий формат времени. Зависит от текущих значений NLS_TERRITORY и NLS_LANGUAGE. Может использоваться с DL или DS, но не с другими элементами |
| TZD | Сокращенное название часового пояса — например, EST или PST. Формат используется только для ввода, что выглядит немного странно |
| TZH | Смещение часового пояса в часах; например -5 означает часовой пояс на 5 часов ранее UTC |
| TZM | Смещение часового пояса в минутах; например -5:30 означает часовой пояс на 5 часов 30 минут ранее UTC (несколько таких часовых поясов существуют) |
| TZR | Регион часового пояса. Например, «US/Eastern» обозначает регион, в котором действительно время EST (Eastern Standard Time) и EDT (Eastern Daylight Time) |
| W | Неделя месяца, от 1 до 5. Неделя 1 начинается с первого дня месяца и заканчивается на седьмой день. Только для вывода |
| WW | Неделя года, от 1 до 53. Только для вывода |
| X | Локальный разделитель дробной части. В американском английском используется символ «точка» (.). Элемент может размещаться перед FF для правильной интерпретации дробных секунд |
| Y,YYY | Год из четырех цифр с запятой |
| YYY или YY или Y | Последние три, две или одна цифры года. При преобразовании символьной строки в дату по умолчанию используется текущий век |

Когда формат даты возвращает текстовое значение (слова вместо чисел), язык, используемый для записи этого текста, определяется параметрами Globalization Support (ранее National Language Support) NLS_DATE_LANGUAGE и NLS_LANGUAGE, или необязательным аргументом языка, передаваемым TO_CHAR и TO_DATE.

Несколько примеров масок формата данных, составленных из предыдущих форматных элементов:

```
'Month DD, YYYY'
'MM/DD/YY Day A.M.'
'Year Month Day HH24:MI:SS'
'J'
'SSSSS-YYYY-MM-DD'
'"A beautiful summer morning on the" DDth" day of "Month"
```

Форматные элементы могут использоваться в любых комбинациях и любом порядке. Старые версии Oracle позволяли указать один элемент даты дважды. Например, в маске "Mon (MM) DD, YYYY" месяц указывается дважды. Однако теперь элемент в форматной маске может указываться только один раз. Например, можно указать только один из элементов MONTH, MON и MM, потому что все три обозначают месяц.

За дополнительными примерами масок формата даты и их использования обращайтесь к описаниям функций TO_CHAR и TO_DATE в главе 10.

ДАТЫ ISO

Элементы IYY и IW представляют год и неделю в стандарте ISO (International Standards Organization). Календарь ISO является классическим примером проектирования «комитетного проектирования» (по принципу «у семи нянек...»). Первым днем года ISO всегда является понедельник, и он определяется по следующим правилам:

- Если 1 января выпадает на понедельник, то год ISO начинается в тот же день.
- Если 1 января выпадает на дни недели от вторника до четверга, то год ISO начинается в предыдущий понедельник.
- Если 1 января выпадает на дни недели с пятницы по субботу, то год ISO начинается со следующего понедельника.

Эти правила порой создают странные ситуации. Например, 31 декабря 2008 года считается первым днем 2009 года по стандарту ISO, но при выводе этой даты в формате IYYY вы получите 31 декабря 2009 года.

Недели ISO всегда начинаются с понедельника и нумеруются с первого понедельника года ISO.

С. Фейерштейн, Б. Прибыл
Oracle PL/SQL. Для профессионалов
6-е издание

Перевел с английского Е. Матвеев

Заведующий редакцией

П. Щеголев

Ведущий редактор

Ю. Сергиенко

Корректоры

С. Беляева, В. Листова

Художественный редактор

В. Шимкевич

Верстка

Л. Родионова

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,

58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 24.10.14. Формат 70х100/16. Усл. п. л. 82,560. Тираж 1000. Заказ

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в Чеховский Печатный Двор. 142300, Чехов, Московская область, г. Чехов, ул. Полиграфистов, д.1.