



Практика реактивного программирования в **Spring 5**

Современному бизнесу необходимы программные системы нового типа, способные оставаться отзывчивыми при любых нагрузках. Эту потребность можно удовлетворить с использованием приемов реактивного программирования; однако разработка таких систем – сложная задача, требующая глубокого понимания предметной области. Для разработки отзывчивых систем разработчики Spring Framework придумали и создали проект Project Reactor.

Данная книга начинается с основ реактивного программирования в Spring. Вы исследуете многочисленные возможности построения эффективных реактивных систем с помощью Spring 5 и других инструментов, таких как WebFlux и Spring Boot. Познакомитесь с методами реактивного программирования и научитесь использовать их для взаимодействий с базами данных и между серверами. Освойте навыки масштабирования с Spring Cloud Streams и научитесь создавать независимые и высокопроизводительные реактивные микросервисы.

С помощью этой книги вы:

- откроете разницу между реактивной системой и реактивным программированием;
- исследуете преимущества реактивных систем и область их применения;
- освоите приемы реактивного программирования в Spring 5;
- получите представление о Project Reactor;
- постройте реактивную систему с использованием Spring 5 и Project Reactor;
- создадите высокоэффективный реактивный микросервис с использованием Spring Cloud;
- научитесь тестировать, выпускать и осуществлять мониторинг реактивных приложений.

Издание адресовано разработчикам на Java, использующим фреймворк Spring для своих приложений и желающих научиться создавать надежные и реактивные приложения, способные автоматически масштабироваться в облаке. Знание основ распределенных систем и асинхронного программирования поможет вам лучше понять темы, охватываемые книгой.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@aliens-kniga.ru

Packt

DMK
ИЗДАТЕЛЬСТВО
www.dmk.ru

ISBN 978-5-97060-747-3



9 785970 607473 >

Практика реактивного программирования в Spring 5

Олег Докука
Игорь Лозинский

Практика реактивного программирования



в Spring 5

DMK
ИЗДАТЕЛЬСТВО

Олег Докука
Игорь Лозинский

Практика реактивного программирования в Spring 5

Создание облачных реактивных систем
с помощью Spring 5 и Project Reactor

Hands-On Reactive Programming in Spring 5

Build cloud-ready, reactive systems
with Spring 5 and Project Reactor

Oleh Dokuka
Igor Lozynskyi

BIRMINGHAM • MUMBAI

Packt>

Практика реактивного программирования в Spring 5

Создание облачных реактивных систем
с помощью Spring 5 и Project Reactor

Олег Докука
Игорь Лозинский

Москва, 2019



УДК 004.432
ББК 32.972.1
Д63

Д63 Олег Докука, Игорь Лозинский
Практика реактивного программирования в Spring 5. –
М.: ДМК Пресс, 2019. – 508 с.

ISBN 978-5-97060-747-3

Данная книга посвящена реактивному программированию в Spring. Описаны многочисленные возможности построения эффективных реактивных систем с помощью Spring 5 и других инструментов, таких как WebFlux, Spring Boot и Project Reactor. Приведены методы реактивного программирования и их использование для взаимодействий с базами данных и между серверами. Рассмотрено создание независимых и высокопроизводительных микросервисов с помощью Spring Cloud Streams.

Издание предназначено разработчикам на Java, использующим фреймворк Spring для своих задач и желающим научиться создавать надежные и реактивные приложения, способные автоматически масштабироваться в облаке.

УДК 004.432
ББК 32.972.1

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-97060-747-3 (рус.) © Олег Докука, Игорь Лозинский, 2018
ISBN 978-1-78728-495-1 (анг.) © Оформление, издание, ДМК Пресс, 2019

В память о моем отце, Иване.

– Игорь Лозинский



Март – это цифровая онлайн-библиотека, которая откроет вам полный доступ к более чем 5 тыс. книг и видеороликов, а также к передовым инструментам, которые помогут вам планировать свое личное развитие и продвигаться по карьерной лестнице. За более подробной информацией обращайтесь на наш веб-сайт www.mart.io.

Что дает подписка?

- Электронные книги и видеоуроки практической направленности, созданные более чем 4 тыс. профессионалами в своем деле, помогут вам меньше времени тратить на обучение и больше на программирование.
- Воспользовавшись инструментом Skill Plans, вы сможете составить индивидуальный план своего обучения.
- Станете получать доступ к новым электронным книгам и видеоурокам каждый месяц.
- У вас будет возможность полнотекстового поиска в библиотеке Март.
- Вы сможете копировать и распечатывать содержимое книг, а также оставлять закладки.

www.PacktPub.com

Знаете ли вы, что издательство Packt предлагает электронные версии всех выпускаемых им книг в форматах PDF и ePub? Приобрести электронные версии можно на сайте www.PacktPub.com, а при покупке печатной книги вы получите скидку на ее электронную копию. За дополнительной информацией обращайтесь по адресу: customercare@packtpub.com.

На сайте www.PacktPub.com можно также найти множество технических статей, подписаться на бесплатные новостные письма и получить исключительные скидки на печатные и электронные книги издательства Packt.

Предисловие

Наконец реактивное программирование получило заслуженную поддержку в таких известных в мире Java продуктах, как Spring Boot и Spring Framework. Как бы вы охарактеризовали решения Spring? Многие пользователи обычно отвечают на этот вопрос коротко: прагматичность. Предлагаемая поддержка реактивного программирования не является исключением, и команда решила продолжать оказывать содействие как реактивным, так и нереактивным стекам. С выбором приходит ответственность, поэтому важно понимать, когда лучше использовать парадигму «реактивного программирования» и какие передовые методы можно применить при создании своей следующей промышленной системы.

Проект Spring позиционируется как поставщик лучших инструментов для разработки любых видов микросервисов. Поддержка реактивного стека Spring помогает разработчикам создавать невероятно эффективные, высокодоступные и надежные конечные точки. Кроме того, реактивные микросервисы на основе Spring терпимо относятся к задержкам в Сети и в меньшей степени подвержены разрушительному влиянию отказов. Только представьте – это решение можно использовать и для Edge API, и для серверной части мобильного приложения, и для тесно взаимосвязанных микросервисов! Не знали? Реактивные микросервисы изолируют медленные транзакции и вознаграждают быстрые.

Как только вы определитесь с потребностями, Project Reactor станет для вас реактивной основой, естественным образом сочетающейся с вашим реактивным проектом на основе Spring. В последних выпусках 3.x этой библиотеки было реализовано большинство реактивных расширений, описанных компанией Microsoft в 2011 году. Наряду со стандартным словарем библиотека Reactor добавляет великолепную поддержку механизма Reactive Streams управления потоком на каждом функциональном шаге и уникальные возможности, такие как передача контекста.

В своей книге Олег и Игорь на примерах пусть и искусственных, но не упрощенных, описывают фантастическое путешествие по миру реактивного программирования и реактивных систем. После краткого введения в библиотеку Project Reactor и историю ее развития вы быстро погрузитесь в исследование практических примеров на основе Spring Boot 2. Кроме того, авторы не упустили случая всерьез охватить тему тестирования и четко и ясно показать, как создается качественный реактивный код.

Авторы предлагают читателям исчерпывающее введение в шаблоны реактивного проектирования для эффективного масштабирования, отвечающего настоящим и будущим потребностям. Они не просто описывают приемы реактивного программирования, но также дают массу практических советов по использованию Spring Boot и Spring Framework. В главе, посвященной перспективам реактивного программирования, авторы разжигают любопытство читателя, знакомя его с некоторыми подробностями о реактивных взаимодействиях с использованием

RSocket – многообещающей технологии, призванной привнести преимущества реактивного программирования в транспортный уровень.

Надеюсь, вы получите от чтения книги столько же удовольствия, сколько и я, и продолжите исследовать новые способы разработки приложений.

Стефан Мальдини (Stéphane Maldini)
Ведущий разработчик, Project Reactor

Об авторах

Олег Докука (Oleh Dokuka) – опытный инженер-программист, обладатель награды Pivotal Champion и один из основных вкладчиков в развитие Project Reactor и Spring Framework. Он хорошо знает, как устроены оба фреймворка, и ежедневно популяризирует идеи реактивного программирования с использованием Project Reactor. Наряду с этим Олег использует Spring Framework и Project Reactor в разработке программного обеспечения, поэтому он не понаслышке знает, как создавать реактивные системы с применением этих технологий.

Игорь Лозинский (Igor Lozynskiy) – старший Java-разработчик, в основном создающий надежные, масштабируемые и невероятно быстрые системы. Имеет за плечами более чем семилетний опыт работы с платформой Java. Увлекается интересными и динамичными проектами как в своей жизни, так и в разработке программного обеспечения.

Об обозревателях

Николай Алименко (Mikalai Alimenkou) – старший менеджер по работе с корпоративными клиентами, технический лидер Java и опытный преподаватель. Эксперт в разработке на Java, в области масштабируемых архитектур, а также методов гибкой разработки, процессов обеспечения качества и управления проектами. Имеет более чем 14-летний опыт разработки программного обеспечения, специализируется на создании сложных, распределенных и масштабируемых систем и на глубоком преобразовании компаний. Активный участник многих международных конференций. Основатель, тренер и независимый консультант тренинг-центра XP Injection. Организатор и идеолог международных конференций Selenium Camp, JEEConf и XP Days Ukraine, а также клуба анонимных разработчиков Anonymous Developers Club (UADEVCLUB).

Назарий Черкас (Nazarii Cherkas) трудится архитектором решений в Hazelcast – в компании, разрабатывающей проекты с открытым исходным кодом, такие как Hazelcast IMDG и Hazelcast Jet. Имеет многолетний опыт деятельности на разных должностях – от инженера-программиста на Java до руководителя группы разработчиков. Принимал участие в различных проектах для множества отраслей – от телекоммуникации и здравоохранения до критических систем, обслуживающих инфраструктуру одной из крупнейших в мире авиакомпаний. Имеет степень магистра информатики Черновицкого национального университета имени Юрия Федьковича.

Томаш Нуркевич (Tomasz Nurkiewicz) – обладатель титула Java Champion. Потратил на программирование половину жизни. Трудится в сфере электронной коммерции. Занимается разработкой свободного программного обеспечения, является самым ценным блогером DZone и очень активно участвует в работе сайта Stack Overflow. Автор, тренер, докладчик, технический обозреватель и легкоатлет. Считает, что не бывает кода, который не тестируется автоматически. Написал книгу о RxJava.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны. Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма. Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги. Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Ракст очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции. Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы. Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Оглавление

Предисловие	7
Глава 1. Причины выбора Spring	22
Основные преимущества реактивности	22
Взаимодействия на основе обмена сообщениями	25
Примеры использования реактивности	30
Причины добавления поддержки реактивности в Spring	33
Реактивность на уровне служб	34
Заключение	42
Глава 2. Реактивное программирование в Spring. Основные понятия	44
Первые реактивные решения в Spring	44
Шаблон «Наблюдатель»	45
Примеры использования шаблона «Наблюдатель»	49
Шаблон «Публикация/Подписка» с использованием @EventListener	52
Создание приложений с @EventListener	54
Создание приложения на основе Spring	54
Реализация бизнес-логики	55
Асинхронные взаимодействия по HTTP с помощью Spring Web MVC	57
Публикация конечной точки SSE	57
Настройка поддержки асинхронного выполнения	59
Создание пользовательского интерфейса с поддержкой SSE	60
Проверка приложения	61
Критический обзор решения	61
RxJava как реактивный фреймворк	62
«Наблюдатель» плюс «Итератор» равно «реактивный поток»	63
Производство и потребление потоков	65
Генерация последовательности асинхронных событий	68
Преобразование потоков и диаграммы Marble	69
Оператор map	69
Оператор filter	70
Оператор count	71
Оператор zip	71
Требования и преимущества RxJava	72
Переделка приложения с RxJava	75
Реализация бизнес-логики	75
Нестандартный SseEmitter	77
Публикация конечной точки SSE	78
Конфигурация приложения	79
Краткая история развития реактивных библиотек	80
Реактивный ландшафт	82
Заключение	84
Глава 3. Reactive Streams – новый стандарт потоков	85
Реактивность для всех	85
Проблема несовместимости API	86

Модели обмена PULL и PUSH	89
Проблема управления потоком данных	95
Медленный производитель и быстрый потребитель	95
Быстрый производитель и медленный потребитель	96
Неограниченная очередь	96
Ограниченная очередь со сбросом избыточных элементов	97
Ограниченная очередь с блокировкой	97
Решение	99
Основные положения стандарта Reactive Streams	99
Требования Reactive Streams в действии	106
Введение в понятие обработчика Processor	109
Проверка совместимости с Reactive Streams	113
Проверка издателя Publisher	115
Проверка подписчика Subscriber	117
JDK 9	121
Асинхронный и параллельный API в Reactive Streams	123
Преобразование реактивного ландшафта	125
Изменения в RxJava	125
Изменения в Vert.x	129
Усовершенствования в Ratpack	130
Драйвер MongoDB с поддержкой Reactive Streams	131
Комбинирование реактивных технологий на практике	132
Заключение	135
Глава 4. Project Reactor – основа реактивных приложений	137
Краткая история Project Reactor	137
Project Reactor 1.x	138
Project Reactor 2.x	141
Основы Project Reactor	142
Добавление библиотеки Reactor в проект	144
Реактивные типы: Flux и Mono	145
Flux	145
Mono	147
Реактивные типы из RxJava 2	148
Observable	148
Flowable	148
Single	148
Maybe	149
Completable	149
Создание последовательностей Flux и Mono	149
Подписка на реактивный поток	151
Реализация своих подписчиков	154
Преобразование реактивных последовательностей с помощью операторов	156
Отображение элементов реактивных последовательностей	157
Фильтрация реактивных последовательностей	158
Сбор данных из реактивных последовательностей	160

Сокращение элементов потока	161
Комбинирование реактивных потоков	164
Пакетная обработка элементов потока	164
Операторы flatMap, concatMap и flatMapSequential	168
Извлечение выборки элементов	170
Преобразование реактивных последовательностей в блокирующие структуры	170
Просмотр элементов при обработке последовательности	171
Материализация и дематериализация сигналов	172
Поиск подходящего оператора	173
Создание потоков данных программным способом	173
Фабричные методы push и create	173
Фабричный метод generate	174
Передача одноразовых ресурсов в реактивные потоки	175
Обертывание транзакций с помощью фабричного метода usingWhen	178
Обработка ошибок	180
Управление обратным давлением	183
Горячие и холодные потоки данных	184
Широковещательная рассылка элементов потока данных	185
Кеширование элементов потока	186
Совместное использование элементов из потока	187
Работа со временем	188
Компоновка и преобразование реактивных потоков	188
Процессоры	190
Тестирование и отладка Project Reactor	191
Дополнения к Reactor	192
Продвинутые средства в Project Reactor	193
Жизненный цикл реактивных потоков данных	193
Этап сборки	193
Этап подписки	195
Выполнение	196
Модель планирования потоков выполнения в Reactor	199
Оператор publishOn	199
Параллельная обработка с помощью publishOn	201
Оператор subscribeOn	202
Оператор parallel	204
Планировщик	204
Контекст	205
Особенности внутренней реализации Project Reactor	209
Макрослияние	210
Микрослияние	211
Заключение	214
Глава 5. Добавление реактивности с помощью Spring Boot 2	216
Быстрый старт как ключ к успеху	217
Использование Spring Roo для ускорения разработки приложений	219
Spring Boot как ключ к созданию быстро растущих приложений	219

Реактивность в Spring Boot 2.0	220
Реактивность в Spring Core	221
Поддержка преобразования реактивных типов	221
Реактивный ввод/вывод	222
Реактивность в Web	224
Реактивность в Spring Data	226
Реактивность в Spring Session	227
Реактивность в Spring Security	228
Реактивность в Spring Cloud	228
Реактивность в Spring Test	229
Реактивность в мониторинге	229
Заключение	230
Глава 6. Неблокирующие и асинхронные взаимодействия с WebFlux	231
WebFlux как основа реактивного сервера	231
Реактивное веб-ядро	234
Реактивные фреймворки Web и MVC	238
Чисто функциональные приемы в WebFlux	242
Неблокирующие взаимодействия между службами с WebClient	246
Реактивный WebSocket API	249
Серверный WebSocket API	250
Клиентский WebSocket API	251
Сравнение WebFlux WebSocket и Spring WebSocket	252
Реактивный поток SSE и легковесная замена WebSocket	253
Реактивные механизмы шаблонов	255
Реактивная безопасность	258
Реактивный доступ к SecurityContext	258
Использование реактивной безопасности	261
Взаимодействия с другими реактивными библиотеками	262
Сравнение WebFlux и Web MVC	263
Законы сравнения фреймворков	264
Закон Литтла	264
Закон Амдала	265
Универсальный закон масштабируемости	269
Анализ и сравнение	272
Модели обработки в WebFlux и Web MVC	272
Влияние моделей обработки на пропускную способность и задержку	274
Проблемы модели обработки в WebFlux	282
Потребление памяти разными моделями обработки	285
Влияние модели обработки на удобство	291
Практическое применение WebFlux	292
Системы на основе микросервисов	292
Системы, обслуживающие клиентов с медленными соединениями	294
Потоковые системы или системы реального времени	294
WebFlux в действии	295
Заключение	299

Глава 7. Реактивный доступ к базам данных	301
Модели обработки данных в современном мире	302
Предметно-ориентированное проектирование	302
Хранение данных в эпоху микросервисов	303
Использование хранилищ разного типа	306
База данных как услуга	307
Разделение данных между микросервисами	309
Распределенные транзакции	310
Событийно-ориентированные архитектуры	310
Согласованность в конечном счете	311
Шаблон SAGA	312
Регистрация событий	312
Разделение ответственности на команды и запросы	313
Бесконфликтно реплицируемые типы данных	314
Система обмена сообщениями как хранилище данных	315
Синхронная модель извлечения данных	316
Протокол связи для доступа к базе данных	316
Драйвер базы данных	318
JDBC	319
Управление соединениями	320
Реактивный доступ к базе данных	321
Spring JDBC	322
Spring Data JDBC	323
Добавление реактивности в Spring Data JDBC	326
JPA	326
Добавление реактивности в JPA	327
Spring Data JPA	327
Добавление реактивности в Spring Data JPA	328
Spring Data NoSQL	329
Ограничения синхронной модели	332
Достоинства синхронной модели	333
Реактивный доступ к данным с использованием Spring Data	334
Реактивное хранилище на основе MongoDB	336
Объединение операций с хранилищем	339
Как работают реактивные хранилища	344
Поддержка разбиения на страницы	345
Детали реализации ReactiveMongoRepository	345
Использование ReactiveMongoTemplate	346
Использование реактивных драйверов (MongoDB)	348
Использование асинхронных драйверов (Cassandra)	350
Реактивные транзакции	352
Реактивные транзакции в MongoDB 4	352
Распределенные транзакции с шаблоном SAGA	361
Реактивные коннекторы в Spring Data	361
Реактивный коннектор MongoDB	361
Реактивный коннектор Cassandra	362

Реактивный коннектор Couchbase	362
Реактивный коннектор Redis	363
Ограничения и ожидаемые улучшения	364
Асинхронный доступ к базам данных	365
Реактивное соединение с реляционной базой данных	367
Использование R2DBC вместе с Spring Data R2DBC	369
Преобразование синхронного хранилища в реактивное	370
С помощью библиотеки rxjava2-jdbc	371
Обертывание синхронного CrudRepository	373
Реактивный Spring Data в действии	378
Заключение	382
Глава 8. Масштабирование с Cloud Streams	383
Брокеры сообщений как основа систем, управляемых сообщениями	384
Балансировка нагрузки на стороне сервера	384
Балансировка нагрузки на стороне клиента с Spring Cloud и Ribbon	386
Брокеры сообщений как эластичный и надежный слой для передачи сообщений	392
Рынок брокеров сообщений	396
Spring Cloud Streams как мост в экосистему Spring	397
Реактивное программирование в облаке	406
Spring Cloud Data Flow	407
Модульная организация приложений с Spring Cloud Function	409
Spring Cloud – функция как часть конвейера обработки данных	416
RSocket для реактивной передачи сообщений с низкой задержкой	420
RSocket и Reactor-Netty	421
RSocket в Java	425
RSocket и gRPC	428
RSocket в Spring Framework	430
RSocket в других фреймворках	432
Проект ScaleCube	432
Проект Proteus	433
В заключение о RSocket	433
Заключение	433
Глава 9. Тестирование реактивных приложений	435
Почему реактивные потоки данных сложно тестировать?	435
Тестирование реактивных потоков с помощью StepVerifier	436
Основы StepVerifier	436
Продвинутые приемы тестирования с использованием StepVerifier	440
Виртуальное время	442
Проверка реактивного контекста	445
Тестирование WebFlux	445
Тестирование контроллеров с помощью WebTestClient	446
Тестирование WebSocket	451

Заключение	455
Глава 10. И наконец, выпуск!	456
Важность поддержки идеологии DevOps в приложениях	456
Мониторинг реактивных Spring-приложений	460
Spring Boot Actuator	460
Добавление механизма мониторинга в проект	460
Конечная точка для получения информации о службе	461
Конечная точка для получения информации о работоспособности	463
Конечная точка для получения информации о параметрах работы	466
Конечная точка управления журналированием	467
Другие важные конечные точки	468
Реализация своей конечной точки для Actuator	469
Безопасность конечных точек	470
Micrometer	472
Параметры по умолчанию в Spring Boot	473
Мониторинг реактивных потоков данных	474
Мониторинг потоков в Reactor	474
Мониторинг планировщиков в Reactor	475
Реализация своих параметров Micrometer	477
Распределенная трассировка с Spring Boot Sleuth	478
Пользовательский интерфейс Spring Boot Admin 2.x	480
Развертывание в облаке	482
Развертывание в Amazon Web Services	485
Развертывание в Google Kubernetes Engine	486
Развертывание в Pivotal Cloud Foundry	487
Обнаружение RabbitMQ в PCF	488
Обнаружение MongoDB в PCF	489
Развертывание в PCF без конфигурации с помощью Spring Cloud Data Flow	491
Knative для FaaS на основе Kubernetes и Istio	491
Советы по успешному развертыванию приложений	492
Заключение	493
Указатель	495

Вступление

Реактивные системы всегда откликаются, что и требуется большинству предприятий. Разработка таких систем – сложная задача, требующая глубокого понимания предмета. К счастью, разработчики Spring Framework создали новую, реактивную версию проекта.

В книге «Практика реактивного программирования в Spring 5» вы познакомитесь с увлекательным процессом разработки реактивных систем на основе фреймворка Spring Framework 5.

Книга начинается со знакомства с основами реактивного программирования в Spring. Вы получите представление о возможностях фреймворка и узнаете об основах реактивного программирования. Далее вашему вниманию будут представлены методы реактивного программирования, способы их использования для организации взаимодействия баз данных и серверов. Все эти задачи продемонстрированы на примере реального проекта, что позволит вам попрактиковаться в применении полученных навыков.

А теперь просим всех на борт реактивной революции в Spring 5!

Кому адресована эта книга

Книга адресована разработчикам на Java, использующим фреймворк Spring для своих приложений и желающим получить возможность создавать надежные и реактивные приложения, которые можно масштабировать в облаке. Предполагается базовое знание распределенных систем и асинхронного программирования.

Содержание книги

Глава 1 «*Причины выбора Spring*» описывает случаи, для которых подходит реактивность. Здесь вы узнаете, чем реактивное решение лучше проактивного, познакомитесь с несколькими примерами кода, демонстрирующими разные способы связи между серверами, а также с потребностями и требованиями бизнеса к современному Spring Framework.

Глава 2 «*Реактивное программирование в Spring – основные понятия*» раскрывает потенциал реактивного программирования, знакомит с основными понятиями на примерах кода. Она демонстрирует мощь реактивного, асинхронного, неблокирующего программирования в Spring Framework на примерах кода и как оно применяется на практике. Здесь вы познакомитесь с моделью «издатель-подписчик», с реактивными событиями Flow и особенностями применения этих методов на практике.

Глава 3 «*Reactive Streams – новый стандарт потоков*» описывает проблемы реактивных расширений Reactive Extensions. На примерах кода раскрывается природа проблем, исследуются разные подходы к их решению. Глава также знакомит со спецификацией Reactive Streams, которая вводит новые компоненты в хорошо известную модель «издатель-подписчик».

Глава 4 «*Project Reactor – основа реактивных приложений*» рассматривает реализацию реактивной библиотеки, полностью осуществляющей спецификацию Reactive Streams. Сначала описываются преимущества Reactor, а затем рассматриваются причины, побудившие разработчиков Spring заняться созданием нового решения. Кроме того, глава знакомит с основами использования данной впечатляющей библиотеки – здесь вы получите представление о Mono и Flux, а также об особенностях применения реактивных типов.

Глава 5 «*Добавление реактивности с помощью Spring Boot 2*» знакомит с реактивными модулями, имеющимися в Spring, которые пригодятся при разработке реактивных приложений. Здесь вы узнаете, как начать использовать модули и как Spring Boot 2 помогает разработчикам быстро настраивать приложения.

Глава 6 «*Неблокирующие и асинхронные взаимодействия с WebFlux*» описывает один из основных модулей – Spring WebFlux, являющийся важным инструментом для организации асинхронного и неблокирующего взаимодействия пользователя и внешних служб. В ней дается обзор преимуществ этого модуля и сравнение его с Spring MVC.

Глава 7 «*Реактивный доступ к базам данных*» описывает модель реактивного программирования доступа к данным на основе Spring 5. Основное внимание уделяется поддержке реактивности в модулях Spring Data, исследуются средства, входящие в состав Spring 5, Reactive Streams и Project Reactor. Здесь представлен код, демонстрирующий реактивный подход к взаимодействию разных баз данных, таких как SQL и NoSQL.

Глава 8 «*Масштабирование с Cloud Streams*» познакомит с реактивными возможностями Spring Cloud Streams. Для начала вашему вниманию будет предложен обзор проблем и недостатков, с которыми можно столкнуться при организации масштабирования на разных серверах. Глава раскроет возможности решения Spring Cloud и продемонстрирует его реализацию на примерах кода с соответствующей конфигурацией Spring Boot 2.

Глава 9 «*Тестирование реактивных приложений*» описывает основы тестирования реактивных приложений. Вы познакомитесь с модулями Spring 5 Test и Project Reactor Test, узнаете, как управлять частотой следования событий, перемещать временные интервалы, расширять пулы потоков выполнения, проверять результаты и переданные сообщения.

Глава 10 «И наконец, выпуск!» содержит подробное пошаговое руководство по развертыванию и мониторингу решения. Наглядно показано, как осуществляется мониторинг реактивных микросервисов с использованием модулей Spring 5. Также в этой главе рассматриваются инструменты, которые пригодятся для сбора данных мониторинга и их отображения.

Что потребуется для работы с книгой

Разработка реактивных систем – сложная задача, требующая глубокого понимания предметной области, поэтому вам потребуется познакомиться с распределенными системами и асинхронным программированием.

Загрузка исходного кода примеров

Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.dmk.ru в разделе «Читателям – Файлы к книгам».

Кроме того, примеры кода к книге доступны на сайте GitHub: <https://github.com/PacktPublishing/Hands-On-Reactive-Programming-in-Spring-5>.

Загрузка цветных иллюстраций

Мы также подготовили документ PDF, содержащий цветные иллюстрации со скриншотами и диаграммами, используемыми в книге. Его можно взять здесь: https://www.packtpub.com/sites/default/files/downloads/9781787284951_ColorImages.pdf.

Типографские соглашения

В книге используется несколько разных стилей оформления текста с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, адреса URL, ввод пользователя и ссылки в Twitter оформляются так, как показано в следующем предложении: «При первом обращении будет вызван обработчик `onSubscribe()`, который сохранит подписку `Subscription` локально и известит издателя `Publisher` о готовности подписчика принимать события через метод `request()`».

Блоки программного кода оформляются следующим образом:


```
@Override
public long maxElementsFromPublisher() {
    return 1;
}
```

Ввод или вывод в командной строке оформляются так:

```
./gradlew clean build
```

Новые термины и важные определения выделяются в тексте полужирным.



Так оформляются предупреждения и важные примечания.



Так оформляются советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезными.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу: dmkpress@gmail.com, при этом укажите название книги в теме письма.

Если есть тема, в которой у вас высокая квалификация, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу: dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг (в тексте или в коде), мы будем благодарны, если вы сообщите нам о ней. Этим вы поможете улучшить последующие версии книги.

Если найдете ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу: dmkpress@gmail.com, и мы исправим их в следующих изданиях.

Нарушение авторских прав

Пиратство в интернете по-прежнему насущная проблема. Издательства «ДМК Пресс» и Рaskt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com и пришлите ссылки на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, способствующую предоставлению качественных материалов.

Глава 1

Причины выбора Spring

В этой главе мы объясним понятие **реактивности** и расскажем, почему реактивные подходы лучше традиционных, для чего рассмотрим примеры, когда традиционные подходы терпят неудачу. Затем исследуем фундаментальные принципы построения надежных систем, которые в большинстве своем являются **реактивными системами**. Узнаем, каковы основные причины, объясняющие необходимость использования механизмов рассылки сообщений для организации взаимодействий между распределенными серверами. Мы покажем, в какие случаи реактивность вписывается как нельзя лучше, расскажем о приемах **реактивного программирования** для создания модульной реактивной системы. Кроме того, обсудим, почему команда разработчиков Spring Framework решила включить реактивный подход в ядро фреймворка **Spring Framework 5**. Прочитав главу, вы поймете важность реактивности и то, почему стоит перенести свои проекты в реактивный мир.

Здесь рассматриваются следующие темы:

- основные преимущества реактивности;
- основные принципы создания реактивных систем;
- случаи, когда реактивный дизайн подходит лучше всего;
- приемы программирования реактивных систем;
- причины включения поддержки реактивности в Spring Framework.

Основные преимущества реактивности

В наши дни стало модным использовать слово **реактивный** – оно такое волнующее и непонятное. Однако стоит ли продолжать популяризовать реактивность, даже после того как слово заняло почетное место на разнообразных международных конференциях? Если забьем в поиск слово «реактивный», то обнаружим, что чаще всего оно встречается в паре со словом «программирование», и вместе они обозначают модель программирования. Однако это не единственный смысл реак-

тивности. За данным словом стоят фундаментальные принципы проектирования, направленные на создание надежных систем. Чтобы понять ценность реактивности как важнейшего принципа проектирования, представим, что мы развиваем малое предприятие.

Допустим, наше малое предприятие – это интернет-магазин, продающий современные товары по привлекательным ценам. Как большинство владельцев подобных магазинов, мы также наняли разработчиков программного обеспечения, которые помогут нам справиться с проблемами. Мы выбрали традиционный подход к разработке программного обеспечения и создали магазин.

Каждый час наш онлайн-магазин обычно посещает 1 тыс. пользователей. Чтобы справиться с потоком покупателей, мы купили современный компьютер и запустили на нем веб-сервер Tomcat, настроив пул с 500 предварительно созданными потоками выполнения. Среднее время отклика на большинство запросов – около 250 мс. Простейшие расчеты показывают, что такая конфигурация позволит нам обслуживать до 2 тыс. запросов в секунду. Согласно статистике, вышеупомянутое число пользователей в среднем производит около 1 тыс. запросов в секунду. Следовательно, текущей производительности системы вполне достаточно для обслуживания средней нагрузки.

Итак, мы настроили приложение с неплохим запасом производительности. Более того, наш интернет-магазин работал вполне стабильно до... последней пятницы ноября, то есть до Черной пятницы.

Черная пятница – важный день и для покупателей, и для продавцов. Покупатели получают возможность купить товар со скидкой, а продавцы – получить дополнительную прибыль. Однако этот день характеризуется необычным наплывом клиентов, что может стать причиной сбоев в работе интернет-магазина.

И конечно же, мы потерпели сокрушительное фиаско! В какой-то момент нагрузка превысила все наши ожидания. В пуле не оказалось свободных потоков выполнения для обработки запросов. Сервер резервного копирования не справился с наплывом покупателей, время отклика возросло, периодически наблюдались сбои. Мы начали терять некоторые запросы, в результате наши клиенты, недовольные долгим обслуживанием, переметнулись к конкурентам.

В итоге мы потеряли большое число клиентов, остались без дополнительной прибыли, а рейтинг магазина рухнул. Все это стало результатом увеличения времени отклика в условиях возросшей нагрузки.

Но не волнуйтесь, это далеко не новая проблема. Даже такие гиганты, как Amazon и Walmart, сталкивались с ней и давно нашли способы ее решения. Но не будем спешить и пройдем тот же путь, каким следовали наши предшественники, чтобы понять основные принципы проектирования надежных систем и дать им общее определение.



Узнать больше о проблемах магазинов-гигантов можно на следующих сайтах:

- Amazon.com. Проблема с отключениями (<https://www.cnet.com/news/amazon-com-hit-with-outages/>);
- Amazon.com. Как отказы приводили к потерям до 66 240 долларов в минуту (<https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#3fd8db37495c>);
- Walmart. Провал в Черную пятницу: веб-сайт не справился с нагрузкой (<https://techcrunch.com/2011/11/25/walmart-black-friday/>).

Теперь главный вопрос: что с этим делать? Из примера выше следует, что приложение должно как-то реагировать на изменение нагрузки и доступности внешних служб. Иначе говоря, оно должно активно реагировать на любые изменения, которые могут ухудшить доступность системы и ее способность откликаться на запросы пользователей.

Один из путей к главной цели – увеличение **эластичности**. Под этим термином понимается способность сохранять отзывчивость при различной рабочей нагрузке, то есть пропускная способность системы должна автоматически увеличиваться с ростом числа пользователей и уменьшаться – со снижением спроса. Эта особенность улучшает отзывчивость системы, потому что в любой момент пропускная способность системы может вырасти и обеспечить приемлемое среднее время задержки.



Время задержки – важная характеристика отзывчивости. При отсутствии должной эластичности из-за роста нагрузки увеличится время задержки, которое напрямую влияет на отзывчивость системы.

Например, увеличить пропускную способность системы можно, расширяя вычислительные мощности или запуская дополнительные экземпляры. В результате возрастет отзывчивость системы. С другой стороны, если поток пользователей уменьшился, система в ответ должна снизить потребление ресурсов, сократив тем самым накладные расходы. Добиться желаемой эластичности можно путем масштабирования – горизонтального или вертикального. Однако масштабирование распределенной системы – сложная задача. Обычно ограничиваются узкими местами или точками синхронизации в системе. С теоретической и практической точек зрения эти проблемы объясняются законом Амдала и универсальной моделью масштабирования Гюнтера Нейла. Мы обсудим их позже – в главе 6 «*Неблокирующие и асинхронные взаимодействия с WebFlux*».



Здесь под накладными расходами понимается стоимость развертывания новых экземпляров в облаке или дополнительное потребление электроэнергии в случае установки добавочных компьютеров.

Однако построение масштабируемой распределенной системы без возможности оставаться отзывчивой независимо от отказов – сложная задача. Представьте ситуацию: какая-то часть системы вдруг оказывается недоступной. Допустим, отказала внешняя платежная система и любые попытки пользователя произвести оплату терпят неудачу. Это нарушит отзывчивость системы, что в некоторых случаях совершенно неприемлемо. Например, если пользователи не смогут с легкостью совершать покупки, они наверняка сбегут в интернет-магазин конкурента.

Чтобы качественно обслуживать клиентов, мы должны позаботиться об отзывчивости системы. Критерием приемлемости является способность системы оставаться отзывчивой в случае отказов или, говоря другими словами, сохранять устойчивость. Этого можно добиться путем изоляции функциональных компонентов системы, помогающей отделить внутренние сбои и обеспечить независимость.

Вернемся к интернет-магазину Amazon. В нем имеется большое число разных функциональных компонентов, отвечающих, например, за вывод списка заказов, оплату, рекламу, прием отзывов от пользователей и др. Например, в случае выхода из строя платежной системы мы можем принять заказ пользователя и запланировать автоматическое повторение запроса, защитив пользователя от сбоев. Другой способ – изоляция от службы приема отзывов пользователей. Если данная служба окажется недоступной, это никак не должно сказаться на возможности оформлять заказы и делать покупки.

Также важно отметить, что эластичность и устойчивость тесно связаны между собой. Получить по-настоящему отзывчивую систему можно, только уделив должное внимание обоим параметрам. Масштабируемость позволяет иметь несколько реплик компонента, чтобы в случае сбоя в одной можно было быстро переключиться на другую и таким способом обеспечить бесперебойную работу системы.



Более подробное описание терминологии вы найдете по следующим ссылкам:

- эластичность (<https://www.reactivemanifesto.org/ru/glossary#Elasticity>);
- отказ (<https://www.reactivemanifesto.org/ru/glossary#Failure>);
- изоляция (<https://www.reactivemanifesto.org/ru/glossary#Isolation>);
- компонент (<https://www.reactivemanifesto.org/ru/glossary#Component>).

Взаимодействия на основе обмена сообщениями

Единственное, что пока остается неясным, – это то, как связываются компоненты распределенной системы и в то же время остаются независимыми, изолирован-

ными и простыми для масштабирования. Рассмотрим связь между компонентами по протоколу HTTP. Следующий фрагмент кода, реализующий взаимодействия по HTTP в Spring Framework 4, наглядно демонстрирует эту идею.

```
@RequestMapping("/resource") // (1)
public Object processRequest() {
    RestTemplate template = new RestTemplate(); // (2)

    ExamplesCollection result = template.getForObject( // (3)
        "http://example.com/api/resource2", //
        ExamplesCollection.class //
    ); //
    ... // (4)

    processResultFurther(result); // (5)
}
```

Данный код выполняет следующие действия.

1. Объявляет обработчик запросов с использованием аннотации `@RequestMapping`.
2. Создает экземпляр `RestTemplate` – самого популярного веб-клиента в Spring Framework 4 для организации взаимодействий типа «запрос-ответ» между службами.
3. Конструирует и посылает запрос. Здесь, используя `RestTemplate`, мы конструируем HTTP-запрос и тут же посылаем его. Обратите внимание, что ответ автоматически отображается в Java-объект и возвращается как результат. Тип ответа определяется вторым параметром метода `getForObject`. Кроме того, префикс `getXxxXXXXXX` определяет HTTP-метод, в данном случае GET.
4. Здесь выполняются дополнительные операции, которые были опущены в этом примере для краткости.
5. Производится следующий этап обработки ответа.

В предыдущем примере мы определили обработчик запросов, который вызывается в ответ на получение запроса от пользователя. Он, в свою очередь, посылает дополнительный HTTP-запрос внешней службе, а затем передает его на следующий этап обработки. Несмотря на то что логика работы этого кода выглядит знакомо и понятно, в нем есть некоторые недостатки. Чтобы понять, что не так в примере, рассмотрим, как протекают события во времени.

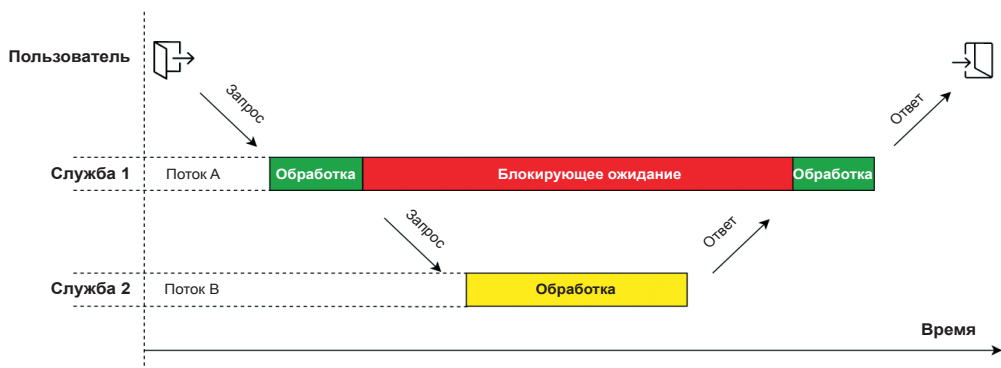


Рис. 1.1. Взаимодействия компонента во времени

Диаграмма отражает фактическое поведение кода в примере выше. Обратите внимание: процессор занят фактической работой только часть времени, тогда как остальное время поток проводит в ожидании завершения операции ввода/вывода и не может использоваться для обслуживания других запросов.



В некоторых языках программирования, таких как C#, Go и Kotlin, этот же код может действовать в неблокирующем режиме при использовании «зеленых» потоков выполнения. Однако в Java такая возможность пока отсутствует, следовательно, поток выполнения фактически будет заблокирован.

С другой стороны, в мире Java имеются пулы потоков выполнения, способные запускать дополнительные потоки. Однако при работе под высокой нагрузкой этот механизм крайне неэффективен и не может использоваться для обработки новых заданий ввода/вывода. Мы еще вернемся к данной проблеме, а также исследуем ее в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux».

Очевидно, что для более эффективного использования ресурсов при выполнении большого количества операций ввода/вывода необходимо задействовать модель асинхронных и неблокирующих взаимодействий. В реальной жизни такой способ взаимодействий организован как обмен сообщениями. Получив сообщение (на телефон или по электронной почте), мы его читаем, а также отвечаем на него, но обычно не ждем ответа и занимаемся другими делами. Безусловно, при таком подходе мы работаем более эффективно и более рационально используем свое время. Взгляните на рис. 1.2.



Более подробное описание терминологии можно найти по ссылкам:

- неблокирующие операции (<https://www.reactivemanifesto.org/ru/glossary#Non-Blocking>);
- ресурс (<https://www.reactivemanifesto.org/ru/glossary#Resource>).

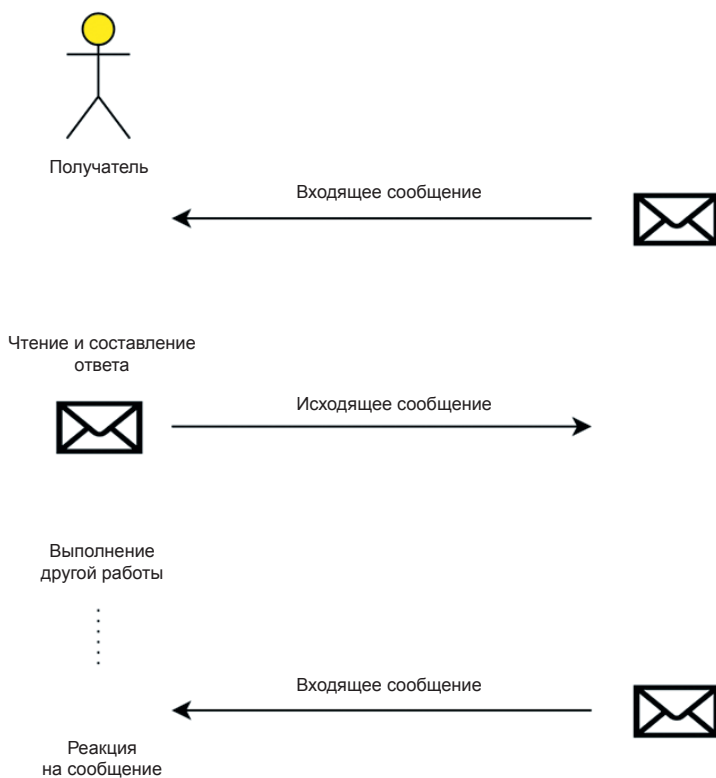


Рис. 1.2. Неблокирующий обмен сообщениями

В целом, чтобы добиться эффективного использования ресурсов при взаимодействии служб в распределенной системе, мы должны взять на вооружение принцип взаимодействий на основе обмена сообщениями. В общих чертах взаимодействие между службами можно описать так: каждый элемент, ожидающий поступления сообщений, реагирует на них при получении, а остальное время пребывает в спящем состоянии, и наоборот, компоненты должны иметь возможность посылать сообщения в неблокирующем режиме. Такой подход к взаимодействиям улучшает масштабируемость системы за счет поддержки независимости от местоположения. Отправляя электронное письмо, мы должны лишь правильно написать электронный адрес получателя, а хлопоты по его доставке на одно из устройств пользователя возьмет на себя почтовый сервер. Это освобождает нас от выбора устройства и дает получателям возможность использовать столько устройств, сколько они пожелают. Кроме того, повышается отказоустойчивость, поскольку отказ одного из устройств не мешает получателю прочитать свою почту с помощью другого устройства.

Один из способов реализации взаимодействий на основе сообщений – использование **брокера сообщений**. В этом случае, осуществляя мониторинг очереди

сообщений, система может управлять эластичностью и нагрузкой. Кроме того, обмен сообщениями делает поток управления более ясным и упрощает общий дизайн. Мы не будем сейчас вдаваться в подробности, потому что наиболее популярные приемы организации взаимодействий на основе сообщений рассматриваются в главе 8 «*Масштабирование с Cloud Streams*».



Фраза **пребывает в спящем состоянии** взята из следующего оригинального документа, который стремится подчеркнуть преимущества взаимодействий на основе обмена сообщениями: <https://www.reactivemaneifesto.org/ru/glossary#Message-Driven>.

Предыдущие утверждения определяют основополагающие принципы построения реактивных систем, как показано на рис. 1.3.

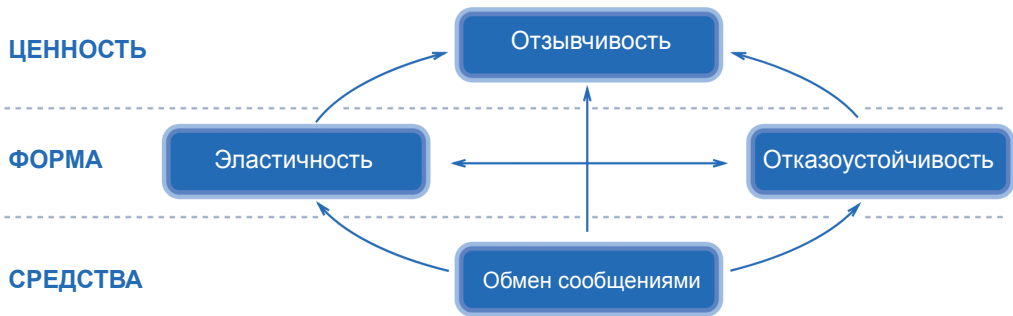


Рис. 1.3. Манифест реактивных систем

На рис. 1.3 мы видим, что главная ценность распределенной системы для любого бизнеса – это отзывчивость. Методы достижения отзывчивости имеют форму эластичности и отказоустойчивости. Наконец, одним из фундаментальных средств обеспечения отзывчивости, эластичности и отказоустойчивости является организация взаимодействий посредством сообщений. Кроме того, системы, построенные на основе этих принципов, просты в сопровождении и легко расширяются, потому что все компоненты системы изолированы и независимы.



Мы не будем подробно обсуждать все понятия, перечисляемые в манифесте реактивных систем, но настоятельно рекомендуем посетить глоссарий: <https://www.reactivemaneifesto.org/ru/glossary/>.

Все эти понятия не новы, их определение дается в манифесте реактивных систем, который является и глоссарием, описывающим реактивные системы. Данный манифест создан для того, чтобы гарантировать одинаковое понимание традиционных идей разработчиками и предпринимателями. Отметим, что реактивные системы и манифест реактивных систем – это архитектурные понятия, они могут применяться и к большим распределенным решениям, и к малым приложениям, выполняющимся на единственном узле.



Важность манифеста реактивных систем (<https://www.reactivemaneifesto.org/ru>) Йонас Бонер (Jonas Bonér), основатель и директор компании Lightbend, объясняет здесь: https://www.lightbend.com/blog/why_do_we_need_a_reactive_manifesto%3F.

Примеры использования реактивности

В предыдущем разделе мы узнали о важности реактивности, об основополагающих принципах реактивных систем, о том, почему организация взаимодействий посредством сообщений является важнейшей составляющей реактивной экосистемы. Чтобы закрепить новые знания, необходимо познакомиться с примерами использования реактивности. Прежде всего под понятием «реактивная система» подразумевается архитектура, которая может применяться где угодно – для реализации простых веб-сайтов, больших корпоративных решений и даже систем потоковой передачи или обработки больших данных. Но начнем с самого простого – рассмотрим пример интернет-магазина, о котором мы уже говорили. Теперь обсудим возможные усовершенствования и изменения в конструкции, которые помогут достичь реактивности. На рис. 1.4 показана общая архитектура предлагаемого решения.

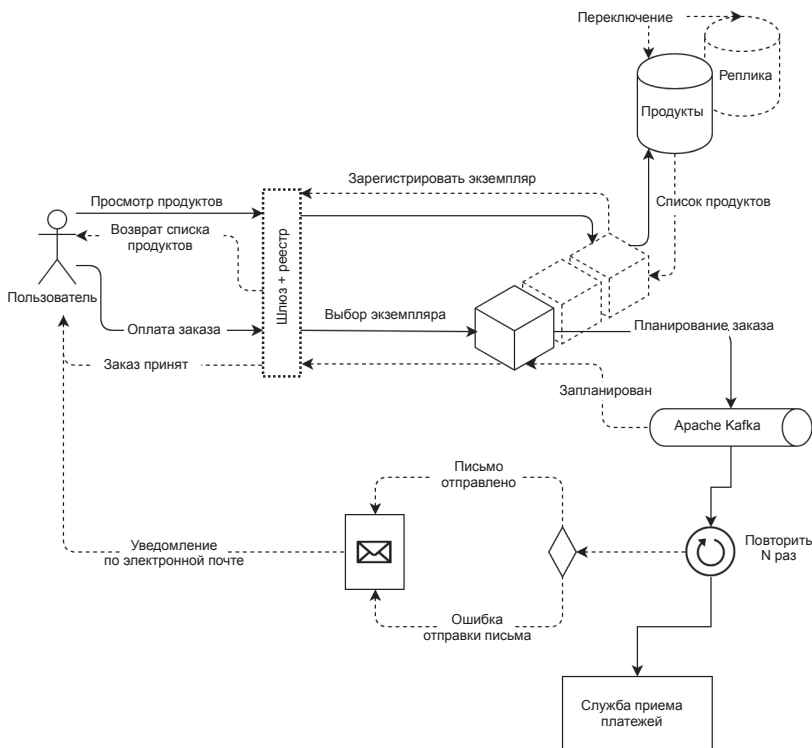


Рис. 1.4. Пример архитектуры интернет-магазина

Диаграмма на рис. 1.4 расширяет список полезных приемов реализации реактивных систем. Здесь мы немного усовершенствовали наш маленький интернет-магазин, применив современный подход на основе микросервисов. В данном случае для обеспечения независимости от местоположения используется шаблон «Шлюз API». Он обеспечивает идентификацию конкретного ресурса, не зная, какие службы отвечают за обработку тех или иных запросов.



Это, однако, означает, что клиент по меньшей мере должен знать название ресурса. Получив название службы в составе URI-запроса, шлюз API может определить конкретный адрес для дальнейшей передачи запроса, обратившись к службе реестра.

Ответственность за поддержание в актуальном состоянии информации о доступных службах возлагается на службу реестра. Отметим, что службы шлюза и реестра в предыдущем примере действуют на одной машине, что может быть полезно для небольших распределенных систем. Кроме того, высокая отзывчивость системы достигается применением репликации к службе. С другой стороны, отказоустойчивость обеспечивается организацией взаимодействий посредством обмена сообщениями с использованием Apache Kafka и независимого прокси для доступа к платежной системе (обозначен на рис. 1.4 точкой с надписью **Повторить N раз**), который отвечает за повторение попыток провести платеж в случае недоступности внешней системы. Также для отказоустойчивости использован прием репликации базы данных на случай, если одна из реплик выйдет из строя. Для достижения высокой отзывчивости мы тут же сообщаем о приеме заказа, асинхронно обрабатываем его и посылаем пользовательскую информацию службе платежей. Окончательное уведомление доставляется позже по одному из поддерживаемых каналов, например по электронной почте. Наконец, этот пример изображает только одну часть системы. В действительности общая диаграмма может быть намного шире и включать более конкретные методы реализации реактивных систем.



Подробнее о принципах проектирования, их достоинствах и недостатках поговорим в главе 8 «Масштабирование с Cloud Streams».

Поближе познакомиться с такими шаблонами, как «Шлюз API», «Служба реестра» и т. д., используемыми для создания распределенных систем, можно на сайте <http://microservices.io/patterns>.

Помимо примера простого интернет-магазина (который тем не менее кому-то может показаться сложным), рассмотрим более замысловатую область, где уместен системный подход. Это аналитика. Под термином «аналитика» подразумевается способность системы обрабатывать гигантские объемы данных, преобразовывать их в процессе работы, держать пользователя в курсе оперативной статистики и т. д.

Представьте, что мы разрабатываем систему мониторинга телекоммуникационной сети и обработки данных сотовой связи. Согласно последнему статистическому отчету, в 2016 году в США действовало 308 334 базовые станции сотовой связи.



Упомянутый статистический отчет доступен на сайте <https://www.statista.com/statistics/185854/monthly-number-of-cell-sites-in-the-united-states-since-june-1986/>.

К сожалению, мы можем лишь приблизительно судить о реальной нагрузке, создаваемой этими базовыми станциями. Однако у нас нет сомнений, что обработка такого огромного объема данных и мониторинг состояния телекоммуникационной сети в реальном времени – действительно сложная задача.

При проектировании данной системы мы можем последовать за одним из эффективных архитектурных методов, который называется **поточковой обработкой данных**. На рис. 1.5 представлена абстрактная архитектура такой потоковой системы.

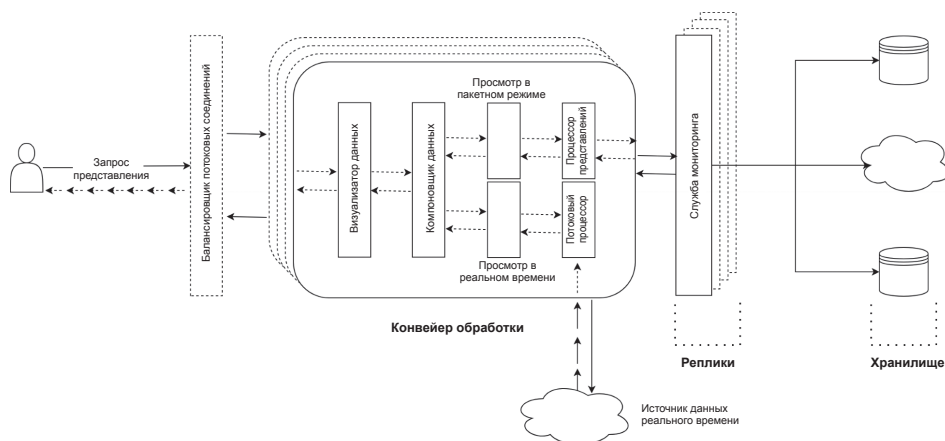


Рис. 1.5. Пример архитектуры системы аналитической обработки данных в режиме реального времени

Рисунок 1.5 демонстрирует, что задача потоковой архитектуры – организация конвейера обработки и преобразования данных. В целом такая система характеризуется низкой задержкой и высокой пропускной способностью. В то же время очень важную роль играет способность откликаться или просто доставлять результаты анализа состояния телекоммуникационной сети. То есть для создания системы с высокой доступностью мы должны взять на вооружение фундаментальные принципы, изложенные в манифесте реактивных систем. Например, большая отказоустойчивость может быть достигнута включением поддержки обратного давления. Под обратным давлением понимается сложный механизм управления распределением рабочей нагрузки между этапами обработки с целью

не допустить перегрузки. Добиться эффективного управления рабочей нагрузкой можно организацией обмена сообщениями через надежного брокера сообщений, который может сохранять их внутри и пересылать по требованию.



Другие приемы поддержки обратного давления рассмотрим в главе 3 «*Reactive Streams – новый стандарт потоков*».

Более того, организовав надлежащее масштабирование каждого компонента, получим возможность гибко изменять пропускную способность системы.



Более подробное описание термина «обратное давление» найдете по ссылке <https://www.reactivemanifesto.org/ru/glossary#Back-Pressure>.

На практике потоки данных могут сохраняться в базах данных и обрабатываться пакетами или частично о в режиме реального времени с применением методов машинного обучения. Как бы то ни было, здесь применимы все фундаментальные принципы манифеста реактивных систем, независимо от предметной области.

Подводя итог, отметим, что принципы построения реактивных систем с успехом используют в массе областей. Их применение не ограничивается предыдущими примерами, поскольку соответствующие принципы можно использовать для создания практически любых распределенных систем, ориентированных на предоставление пользователям эффективной обратной связи.

В следующем разделе рассмотрим причины перехода Spring Framework к применению реактивности.

Причины добавления поддержки реактивности в Spring

В предыдущем разделе мы рассмотрели несколько интересных примеров, где преимущества реактивных подходов проявляются во всей красе. Охватили также такие базовые принципы, как эластичность и устойчивость, и познакомились с системами на основе микросервисов, обычно используемых для создания реактивных систем.

Это дало нам понимание архитектурной перспективы, но ничего не сообщило о реализации. Однако важно подчеркнуть сложность реактивных систем и что конструирование таких систем – непростая задача. Чтобы проще было создавать реактивные системы, мы должны проанализировать фреймворки, пригодные для этого, а затем выбрать какой-нибудь из них. Один из самых популярных способов выбора фреймворка – анализ его особенностей, релевантности и наличие сообщества.

В мире JVM наиболее известными фреймворками для создания реактивных систем являются экосистемы Akka и Vert.x.

С одной стороны, у нас есть Akka – популярный фреймворк с огромным списком возможностей и обширным сообществом. Однако первоначально Akka был частью экосистемы языка Scala и долгое время демонстрировал свою силу только в решениях на Scala. Несмотря на то что Scala является языком JVM, он заметно отличается от Java. Несколько лет назад в Akka была добавлена прямая поддержка Java, но по каким-то причинам он не получил в мире Java такой же популярности, как в Scala.

С другой стороны, существует фреймворк Vert.x, который также является мощным решением для создания эффективных реактивных систем. Vert.x создавался как неблокирующая альтернатива Node.js, ориентированная на события, которая выполняется под управлением виртуальной машины Java. Однако Vert.x вступил в конкурентную борьбу всего несколько лет назад, тогда как последние 15 лет рынок фреймворков для разработки гибких и надежных приложений уверенно удерживает Spring Framework.



Дополнительную информацию о ландшафте инструментов Java вы найдете по ссылке: <https://www.quora.com/Is-it-worth-learning-Java-Spring-MVC-as-of-March-2016/answer/Krishna-Srinivasan-6?srid=xCnf>.

Фреймворк Spring Framework предлагает широкий спектр возможностей для создания веб-приложений с использованием дружественной модели программирования. Однако долгое время он имел некоторые ограничения, мешающие созданию надежных реактивных систем.

Реактивность на уровне служб

К счастью, большой спрос на реактивные системы стал причиной создания нового проекта Spring – **Spring Cloud**. Spring Cloud Framework – это основа для проектов, решающая конкретные проблемы и упрощающая конструирование распределенных систем. Следовательно, экосистема Spring Framework имеет непосредственное отношение к тем, кто создает реактивные системы.



Узнать больше об основных возможностях, компонентах и особенностях этого проекта можно по ссылке <http://projects.spring.io/spring-cloud/>.

Здесь мы не будем обсуждать детали функционирования Spring Cloud Framework, а наиболее важные его части, помогающие в разработке реактивных систем, рассмотрим в главе 8 «*Масштабирование с Cloud Streams*». Тем не менее следует отметить, что это решение помогает с минимальными усилиями создавать надежные реактивные системы на основе микросервисов.

Однако общий дизайн – лишь один из конструктивных элементов реактивной системы. В манифесте реактивных систем отмечается:

«Большие системы состоят из подсистем, имеющих те же свойства, и, следовательно, зависят от их реактивных характеристик. То есть принципы реактивных систем применяются на всех уровнях, что позволяет компоновать их между собой».

Таким образом, очень важно обеспечить реактивный дизайн и реализацию на уровне компонентов. В данном контексте термин «дизайн» относится к отношениям между компонентами и к методам программирования, с помощью которых их объединяют. Наиболее популярной техникой программирования на Java является **императивное программирование**.

Рассмотрим рис. 1.6, чтобы понять, насколько императивное программирование согласуется с принципами организации реактивных систем.

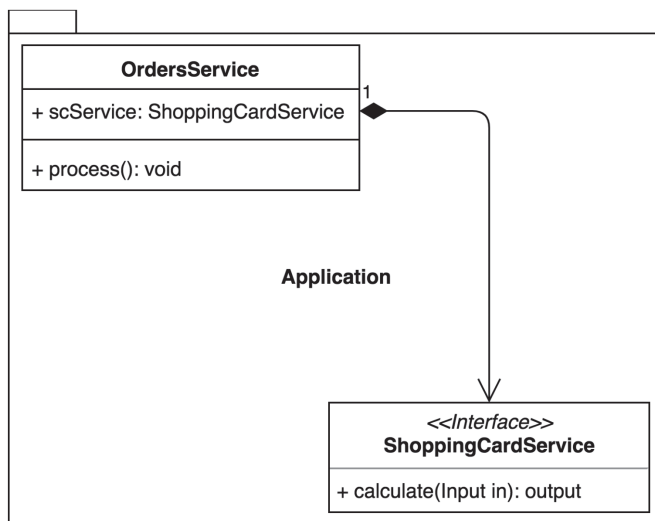


Рис. 1.6. UML-диаграмма отношений между компонентами

Здесь изображены два компонента приложения интернет-магазина. В данном случае OrdersService вызывает ShoppingCartService в процессе обработки запроса пользователя. Допустим, ShoppingCartService выполняет продолжительную операцию ввода/вывода – например, посылает HTTP-запрос или обращается к удаленной базе данных. Чтобы выявить недостатки императивного программирования, рассмотрим следующую распространенную реализацию взаимодействий между этими двумя компонентами.

```

interface ShoppingCartService {                                // (1)
    Output calculate(Input value);                             //
}                                                                //
  
```



```

class OrdersService {                                     // (2)
    private final ShoppingCardService scService;         //

    void process() {                                     //
        Input input = ...;                               //
        Output output = scService.calculate(input);      // (2.1)
        ...                                              // (2.2)
    }                                                    //
}                                                        //

```

Вот как действует этот код.

1. Объявление интерфейса `ShoppingCardService`. Оно соответствует диаграмме классов на рис. 1.6 и определяет единственный метод `calculate`, который принимает один аргумент и возвращает ответ после обработки.
2. Объявление класса `OrderService`. Здесь в строке (2.1) производится синхронный вызов экземпляра `ShoppingCardService` и прием его результата. В строке (2.2) находится остальной код, обрабатывающий полученный результат.
3. В данном случае наши службы оказываются тесно связанными, то есть выполнение `OrderService` сильно зависит от выполнения `ShoppingCardService`. К сожалению, при таком подходе у нас нет возможности осуществлять другие действия, пока `ShoppingCardService` занят обработкой запроса.

Как нетрудно понять из примера, в мире Java выполнение `scService.calculate(input)` блокирует поток `Thread`, в котором выполняется логика `OrdersService`. То есть, чтобы организовать независимую обработку в `OrderService`, мы должны запустить дополнительный поток `Thread`. Как будет показано далее в этой главе, создание нового потока `Thread` может оказаться непоправимым расточительством. Следовательно, с точки зрения реактивной системы такое поведение абсолютно неприемлемо.



Блокирующие взаимодействия прямо противоречат принципу взаимодействий с применением сообщений, который явно предлагает возможность неблокирующих взаимодействий. Дополнительную информацию вы найдете по ссылке <https://www.reactivemanifesto.org/ru#message-driven>.

Впрочем, в Java проблему можно решить, используя для организации между компонентами технику обратных вызовов.

```

interface ShoppingCardService {                         // (1)
    void calculate(Input value, Consumer<Output> c);    //
}                                                        //

class OrdersService {                                   // (2)
    private final ShoppingCardService scService;       //
}                                                        //

```

```

void process() {                                     //
    Input input = ...;                               //
    scService.calculate(input, output -> {           // (2.1)
        ...                                           // (2.2)
    });                                              //
}                                                  //
}                                                  //

```

Вот как действует данный код.

1. Объявление интерфейса `ShoppingCardService`. На этот раз метод `calculate` принимает два аргумента и ничего не возвращает. То есть вызывающий компонент не должен ждать ответа, потому что последний будет передан обратному вызову `Consumer<>` по мере готовности.
2. Объявление `OrderService`. Здесь в строке (2.1) осуществляется асинхронный вызов `ShoppingCardService`, который тут же возвращает управление, и выполнение `OrderService` продолжается. Позднее, когда `ShoppingCardService` передаст ответ функции обратного вызова, мы сможем обработать его (2.2).

Теперь `OrderService` передает функцию обратного вызова, реализующую реакцию на окончание вычислений. Следовательно, отныне `OrderService` не связан с `ShoppingCardService` и, как предполагает реализация метода `ShoppingCardService#calculate`, будет уведомлен о готовности результата посредством обратного вызова, синхронно или асинхронно.

```

class SyncShoppingCardService implements ShoppingCardService { // (1)
    public void calculate(Input value, Consumer<Output> c) {      //
        Output result = new Output();                             //
        c.accept(result);                                          // (1.1)
    }                                                            //
}                                                            //

class AsyncShoppingCardService implements ShoppingCardService { // (2)
    public void calculate(Input value, Consumer<Output> c) {      //
        new Thread(() -> {                                        // (2.1)
            Output result = template.getForObject(...);           // (2.2)
            ...                                                    //
            c.accept(result);                                       // (2.3)
        }).start();                                                // (2.4)
    }                                                            //
}                                                            //

```

Вот как действует этот код.

1. Объявление класса `SyncShoppingCardService`. Данная реализация предполагает отсутствие блокирующих операций. Поскольку ввод/вывод не производится, результат можно вернуть немедленно посредством функции обратного вызова (1.1).

2. Объявление класса `AsyncShoppingCardService`. В данном случае имеет место блокирующая операция ввода/вывода в строке (2.2), следовательно, для обработки запроса запускается новый поток `Thread` (2.1) (2.4). После получения результата он передается через функцию обратного вызова.

В данном примере представлена синхронная реализация `ShoppingCardService`, которая остается в рамках синхронного выполнения и не дает никаких преимуществ. Однако в асинхронной версии мы покидаем синхронный мир и выполняем запрос в отдельном потоке `Thread`. Компонент `OrdersService` отделен от процесса обработки и будет уведомлен о завершении через функцию обратного вызова.

Преимущество этого приема в том, что компоненты не связаны друг с другом во времени. То есть после вызова метода `scService.calculate` компонент может тут же продолжить выполнять другие операции, не дожидаясь ответа от `ShoppingCardService`.

А недостаток его в том, что для использования техники обратных вызовов разработчик должен хорошо владеть приемами многопоточного программирования, чтобы не попасть в ловушку изменения общих данных и избежать ада обратных вызовов.



Фактически понятие «ад обратных вызовов» зародилось в JavaScript (<http://callbackhell.com>), но оно применимо и к Java.

К счастью, обратные вызовы не единственный способ организации асинхронных взаимодействий. Другой способ основан на использовании `java.util.concurrent.Future`, который до определенной степени скрывает свое поведение и тоже отделяет компоненты друг от друга.

```
interface ShoppingCardService {                                // (1)
    Future<Output> calculate(Input value);                       //
}                                                                //

class OrdersService {                                          // (2)
    private final ShoppingCardService scService;              //
                                                                //
    void process() {                                           //
        Input input = ...;                                     //
        Future<Output> future = scService.calculate(input);    // (2.1)
        ...                                                    //
        Output output = future.get();                          // (2.2)
        ...                                                    //
    }                                                          //
}                                                                //
```

Вот как действует этот код.

1. Объявление интерфейса `ShoppingCardService`. Здесь метод `calculate` принимает один аргумент и возвращает экземпляр `Future`. `Future` – это класс обертки, позволяющей проверить доступность результата или заблокировать дальнейшее выполнение в его ожидании.
2. Объявление `OrderService`. Здесь в строке (2.1) мы выполняем асинхронный вызов `ShoppingCardService` и получаем экземпляр `Future`. Затем можно продолжать выполнять другие операции, пока не завершится асинхронная обработка запроса. Спустя некоторое время проверим завершение работы `ShoppingCardService#calculation` и получим результат. В данном случае (2.2) попытка может окончиться блокировкой или вернет результат немедленно.

Как видно из предыдущего кода, класс `Future` позволяет отложить получение результата. Благодаря классу `Future` мы избегаем обратных вызовов и абстрагируемся от сложностей многопоточного программирования. В любом случае, чтобы получить нужный результат, мы должны, возможно, на продолжительное время заблокировать текущий поток `Thread` и синхронизироваться с внешним выполнением, что заметно ухудшает масштабируемость.

С целью исправить проблему в Java 8 предлагаются `CompletionStage` и его прямая реализация `CompletableFuture`. Эти классы поддерживают promise-подобные API и позволяют писать такой код.



Узнать больше о механизмах `future` и `promise` отложенных вычислений можно по ссылке https://ru.wikipedia.org/wiki/Futures_and_promises.

```
interface ShoppingCardService {                                // (1)
    CompletionStage<Output> calculate(Input value);           //
}                                                            //

class OrderService {                                          //(2)
    private final ComponentB componentB;                      //
    void process() {                                          //
        Input input = ...;                                    //
        componentB.calculate(input)                          // (2.1)
        .thenApply(out1 -> { ... })                          // (2.2)
        .thenCombine(out2 -> { ... })                        //
        .thenAccept(out3 -> { ... })                         //
    }                                                         //
}                                                            //
```

Вот как действует этот код.

1. Объявление интерфейса `ShoppingCardService`. В данном случае метод `calculate` принимает один аргумент и возвращает экземпляр `CompletionStage`. `CompletionStage` – это класс обертки, похожий на `Future`,

но позволяющий обрабатывать и возвращать результат в декларативной манере функционального программирования.

2. Объявление `OrderService`. Здесь в строке (2.1) мы асинхронно вызываем `ShoppingCardService` и немедленно получаем экземпляр `CompletionStage`. В целом `CompletionStage` действует подобно `Future`, но предлагает более гибкий API с такими методами, как `thenAccept` и `thenCombine`. Они позволяют определить преобразующие операции с результатом и конечных потребителей преобразованного результата.

Благодаря `CompletionStage` можно писать код в функциональном и декларативном стиле, который выглядит прозрачно и обрабатывает результат асинхронно. Кроме того, можно избежать ожидания результата и определить функцию для его обработки, когда он станет доступным. Более того, все предыдущие приемы прошли экспертизу разработчиков Spring и уже реализованы в большинстве проектов, входящих в состав этого фреймворка. К сожалению, несмотря на то что `CompletionStage` предлагает более широкие возможности для создания эффективного и легко читаемого кода, имеют место некоторые упущения. Например, Spring 4 MVC долгое время не поддерживал `CompletionStage` и для тех же целей предлагал собственную реализацию в форме `ListenableFuture`. Причина в том, что разработчики Spring 4 стремились сохранить совместимость со старыми версиями Java. Давайте рассмотрим пример использования `AsyncRestTemplate`, чтобы понять, как работать с классом `ListenableFuture`. Следующий код демонстрирует один из вариантов использования `ListenableFuture` в паре с `AsyncRestTemplate`.

```
AsyncRestTemplate template = new AsyncRestTemplate();
SuccessCallback onSuccess = r -> { ... };
FailureCallback onFailure = e -> { ... };
ListenableFuture<?> response = template.getForEntity(
    "http://example.com/api/examples",
    ExamplesCollection.class
);
response.addCallback(onSuccess, onFailure);
```

Код демонстрирует реализацию асинхронных взаимодействий в стиле обратных вызовов. По сути, этот прием является грубым хаком и за кулисами Spring Framework заворачивает блокирующие сетевые вызовы в отдельные потоки выполнения. Кроме того, Spring MVC опирается на Servlet API, который обязывает все реализации использовать модель выполнения запросов в отдельных потоках.



Многое изменилось с выпуском Spring Framework 5 и нового проекта Reactive WebClient. Теперь благодаря поддержке WebClient появилась возможность использовать неблокирующие методы взаимодействий между службами. Также в Servlet 3.0 теперь есть поддержка асинхронных взаимодействий – клиент-сервер, в Servlet 3.1 добавлена возможность неблокирующей записи в операциях ввода/вывода, и в целом новые асинхронные механизмы в Servlet 3 прекрасно интегрированы в Spring

MVC. Единственная проблема состоит в том, что в Spring MVC отсутствует готовый, асинхронный и неблокирующий клиент, что сводит на нет все преимущества улучшенных сервлетов.

Это далеко не оптимальная модель. Чтобы понять причины неэффективности данного подхода, необходимо принять во внимание дороговизну многопоточно-го выполнения. С одной стороны, многопоточность сама по себе очень сложна. Работая с потоками выполнения, приходится учитывать множество самых разных мелочей, таких как доступ к общей памяти из разных потоков, синхронизация, обработка ошибок и т. д. Дизайн многопоточности в Java, в свою очередь, предполагает, что несколько потоков могут использовать единственный процессор для одновременного выполнения заданий. Так как в этом случае процессорное время распределяется между несколькими потоками, то возникает необходимость **переключения контекста**. То есть, чтобы возобновить выполнение потока позже, нужно сохранить и загрузить регистры, карты памяти и выполнить другие операции с большим объемом вычислений. Из-за этого снижается эффект от применения многопоточности в случаях со значительным количеством потоков и небольшим числом процессоров.



Узнать больше о стоимости переключения контекста можно по ссылке https://ru.wikipedia.org/wiki/Переключение_контекста.

Кроме того, типичный поток выполнения в Java предъявляет довольно серьезные требования к потреблению памяти. Размер стека типичного потока в 64-разрядной версии Java VM равен 1024 Кбайт. С одной стороны, попытка обработать сразу ~64 000 запросов с использованием модели, выделяющей самостоятельный поток для каждого запроса, потребует около 64 Гбайт памяти, что может быть довольно дорого с точки зрения бизнеса. С другой стороны, решение на основе традиционных пулов потоков ограниченного размера и предварительно настроенной очереди запросов менее надежно. Клиентам придется ждать ответа слишком долго.

Для этих целей в манифесте реактивных систем рекомендуется использовать неблокирующие операции, и данная рекомендация была проигнорирована в экосистеме Spring. С другой стороны, отсутствие хорошей интеграции с реактивными серверами, такими как Netty, отчасти обостряет проблему переключения контекста.



Получить больше информации о среднем количестве соединений можно по ссылке <https://stackoverflow.com/questions/2332741/what-is-the-theoretical-maximum-number-of-open-tcp-connections-that-a-modernlin/2332756#2332756>.

Термин «поток» относится к памяти, выделенной для объекта потока и для стека потока. Подробности найдете по ссылке <http://xmlandmore.blogspot.com/2014/09/jdk-8-thread-stack-size-tuning.html?m=1>.

Важно отметить, что асинхронная обработка не ограничивается простым шаблоном запрос/ответ, и иногда приходится иметь дело с бесконечными потоками данных и использовать для их обработки цепочки преобразований с поддержкой обратного давления.

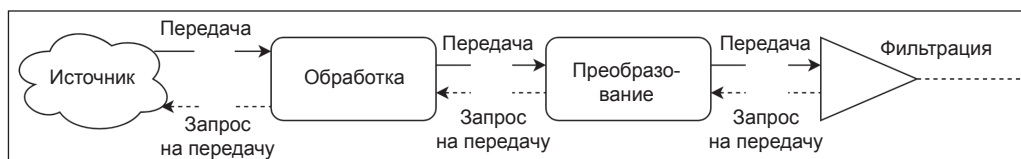


Рис. 1.7. Пример реактивного конвейера

Один из способов справиться с подобной ситуацией – использовать приемы реактивного программирования, в том числе методы асинхронной обработки событий и использование цепочки этапов преобразований. То есть приемы реактивного программирования в полной мере соответствуют требованиям проектирования реактивных систем. В следующих главах мы рассмотрим важность и ценность использования подходов реактивного программирования для создания реактивных систем.

К сожалению, методы реактивного программирования плохо поддерживались в Spring Framework. Это стало еще одним ограничением для разработчиков современных приложений и снизило конкурентоспособность фреймворка. Как следствие все упомянутые пробелы, связанные с реактивными системами и реактивным программированием, просто увеличили потребность в кардинальном улучшении фреймворка. Наконец, добавление поддержки реактивности на всех уровнях способствовало бы увеличению привлекательности Spring Framework и дало разработчикам мощный инструмент разработки реактивных систем. По этим причинам ключевые разработчики фреймворка решили реализовать новые модули, концентрирующие в себе всю мощь Spring Framework как основы для реактивных систем.

Заключение

В этой главе мы рассказали о требованиях к современным и экономически эффективным ИТ-решениям. Мы поведали, почему большие компании, такие как Amazon, потерпели неудачу в попытках использовать старые архитектурные решения в современных облачных окружениях.

Мы также установили, что действительно имеется потребность в новых архитектурных шаблонах и приемах программирования для удовлетворения постоянно растущего спроса на удобные, эффективные и интеллектуальные цифровые услуги. С помощью манифеста реактивных систем разобрали понятие реактивности и показали, как и почему эластичность, устойчивость и взаимодействие с исполь-

зованием сообщений помогают добиться высокой отзывчивости, что является основным нефункциональным требованием к системам в нашу цифровую эпоху. И конечно же, мы привели примеры ситуаций, когда реактивные системы позволяют предприятиям добиться своих целей.

В этой главе подчеркнуты различия между реактивной системой как архитектурным шаблоном и реактивным программированием как подходом к программированию. Мы рассказали, как и почему сочетание этих двух сторон реактивности позволяет создавать высокоэффективные IT-решения.

Для углубленного исследования особенностей Reactive Spring 5 необходимо знать и понимать основы реактивного программирования, изучить основные понятия и шаблоны, поэтому в следующей главе займемся изучением сути реактивного программирования, его истории и ландшафта реактивного программирования в мире Java.

Глава 2

Реактивное программирование в Spring. Основные понятия

В предыдущей главе вы узнали, почему так важно создавать реактивные системы и как реактивное программирование помогает делать это. В данной главе познакомимся с некоторыми инструментами, уже имеющимися в составе Spring Framework, а также с отдельными важными понятиями реактивного программирования, попутно исследуя библиотеку RxJava – самую первую и самую известную реактивную библиотеку в мире Java.

Будут рассмотрены следующие темы:

- шаблон «Наблюдатель»;
- реализация шаблона «Публикация/Подписка» в Spring;
- события, посылаемые сервером;
- история появления RxJava и основные понятия;
- диаграммы Marble;
- примеры использования методов реактивного программирования;
- текущее положение дел в мире реактивных библиотек.

Первые реактивные решения в Spring

В предыдущей главе упоминалось, что существует множество шаблонов и методов программирования, которые можно использовать как строительные блоки для создания реактивных систем. Например, для реализации архитектур на основе обмена сообщениями часто используются обратные вызовы и `CompletableFuture`. Также говорилось, что *реактивное программирование* является главным кандидатом на эту роль. Прежде чем приступить к более детальным исследованиям, же-

лательно было бы осмотреться и найти другие решения, уже использующиеся на протяжении многих лет.

В главе 1 «Причины выбора Spring» мы узнали, что в Spring 4.x появился класс `ListenableFuture`, который наследует Java-класс `Future` и позволяет асинхронно выполнять такие операции, как HTTP-запросы. К сожалению, лишь немногие компоненты Spring 4.x поддерживают новейший класс `CompletableFuture`, появившийся в Java 8, обладающий некоторыми замечательными методами организации асинхронных вычислений.

Тем не менее в Spring Framework имеются другие компоненты, которые очень пригодятся при создании реактивных приложений. Рассмотрим некоторые из них.

Шаблон «Наблюдатель»

Прежде чем двинуться дальше, вспомним один довольно старый и хорошо известный шаблон проектирования – **шаблон «Наблюдатель»** (Observer). Это один из 23 известных шаблонов «Банды четырех» (Gang of Four, GoF). На первый взгляд может показаться, что шаблон «Наблюдатель» не связан с реактивным программированием, однако далее вы увидите, как он с небольшими изменениями определяет основу реактивного программирования.



Прочитать больше о шаблонах проектирования «Банды четырех» можно в книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) «*Design Patterns: Elements of Reusable Object-Oriented Software*» (https://ru.wikipedia.org/wiki/Design_Patterns)¹.

В шаблон «Наблюдатель» входит субъект, который хранит список своих так называемых наблюдателей. Субъект уведомляет наблюдателей обо всех изменениях в своем состоянии, обычно вызывая один из их методов. Этот шаблон играет важную роль при реализации систем, основанных на событиях. Шаблон «Наблюдатель» также является важнейшей частью шаблона «модель – представление – контроллер» (Model-View-Controller, MVC), и почти все библиотеки поддержки пользовательского интерфейса применяют его.

Воспользуемся аналогией из повседневной жизни. Мы можем применить этот шаблон на техническом портале для реализации подписки на рассылку новостей, для чего нужно зарегистрировать свой электронный адрес на интересующем нас сайте. Тот будет присылать уведомления в форме новостных писем, как показано на рис. 2.1.

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Паттерны проектирования. М.: Питер, 2016. – Прим. перев.

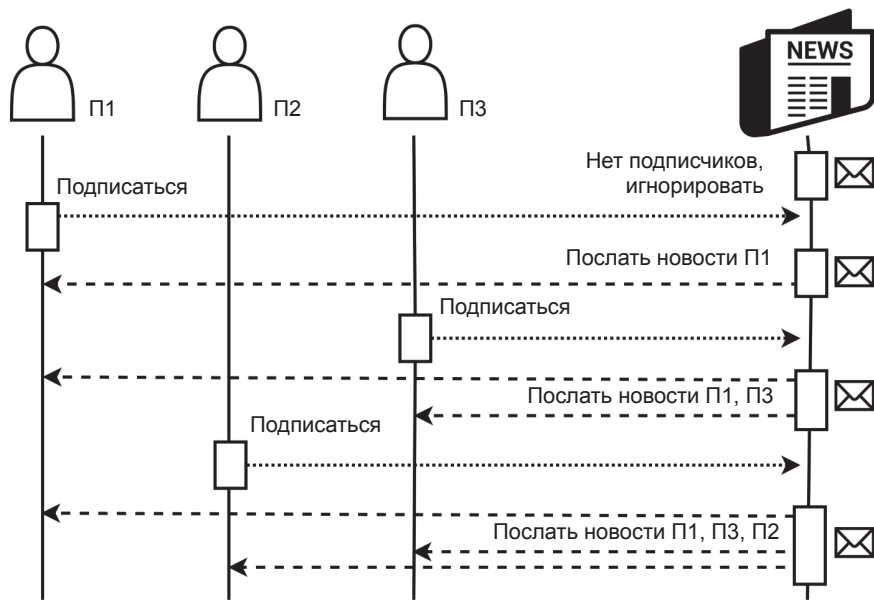


Рис. 2.1. Аналогия шаблона «Наблюдатель» в повседневной жизни: подписка на рассылку новостей

Шаблон «Наблюдатель» позволяет регистрировать зависимости «один ко многим» между объектами в процессе выполнения. В то же время он ничего не знает о тонкостях реализации компонента-подписчика (для безопасности наблюдатель должен знать тип входящего события). Это дает нам возможность отделить части приложения, даже активно взаимодействующие друг с другом. Взаимодействия такого рода обычно являются однонаправленными и помогают эффективно распределять события в системе, как показано на рис. 2.2.

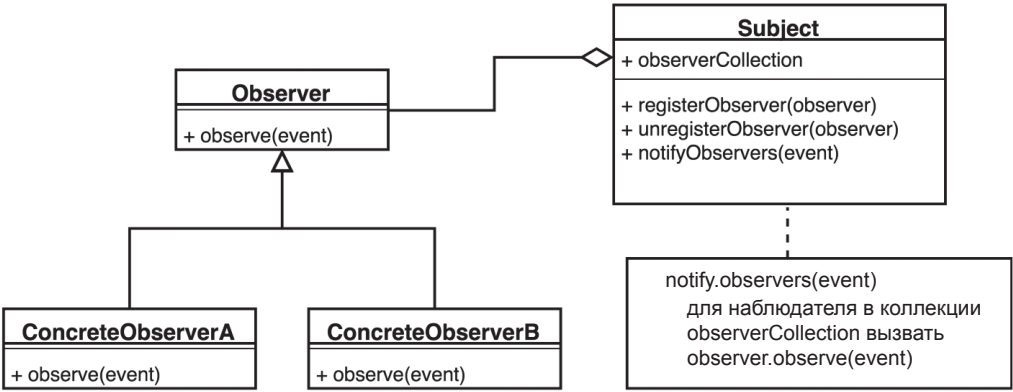


Рис. 2.2. UML-диаграмма класса Observer

Как можно судить по диаграмме на рис. 2.2, типичная реализация шаблона «Наблюдатель» включает два интерфейса, Subject и Observer. Здесь Observer регистрируется в Subject и принимает уведомления от него. Subject может генерировать собственные события или сам вызываться другими компонентами. Определим интерфейс Subject на языке Java.

```
public interface Subject<T> {  
    void registerObserver(Observer<T> observer);  
    void unregisterObserver(Observer<T> observer);  
    void notifyObservers(T event);  
}
```

Этот обобщенный интерфейс параметризуется типом события T, что увеличивает безопасность программного кода. Он также содержит методы для управления подпиской (registerObserver, unregisterObserver и notifyObservers), последний из которых осуществляет рассылку событий. А вот как может выглядеть определение интерфейса Observer.

```
public interface Observer<T> {  
    void observe(T event);  
}
```

Observer – это обобщенный интерфейс, параметризованный типом T. Он определяет единственный метод observe, который реализует обработку событий. Оба компонента, Observer и Subject, знают друг о друге не больше, чем определено в этих интерфейсах. Реализация Observer может нести ответственность за оформление подписки или вообще ничего не знать о существовании Subject. В последнем случае должен существовать некий третий компонент, отвечающий за поиск подходящего экземпляра Subject и регистрацию подписки. Например, данную роль может играть контейнер, поддерживающий механизм внедрения зависимостей (Dependency Injection). Он сканирует путь к классам classpath в поисках всех экземпляров Observer с аннотацией @EventListener и верной сигнатурой, а затем регистрирует найденные компоненты в экземплярах Subject.



Классическим примером контейнера внедрения зависимостей может служить сам Spring Framework. Если эта сторона фреймворка вам неизвестна, прочитайте замечательную статью Мартина Фаулера (Martin Fowler): <https://martinfowler.com/articles/injection.html>.

Теперь реализуем двух простых наблюдателей, которые просто получают сообщения типа String и выводят их.

```
public class ConcreteObserverA implements Observer<String> {  
    @Override  
    public void observe(String event) {  
        System.out.println("Observer A: " + event);  
    }  
}
```

```
    }  
}  
  
public class ConcreteObserverB implements Observer<String> {  
    @Override  
    public void observe(String event) {  
        System.out.println("Observer B: " + event);  
    }  
}
```

Также нам нужно написать реализацию `Subject<String>`, которая будет генерировать события типа `String`.

```
public class ConcreteSubject implements Subject<String> {  
  
    private final Set<Observer<String>> observers =           // (1)  
        new CopyOnWriteArraySet<>();  
  
    public void registerObserver(Observer<String> observer) {  
        observers.add(observer);  
    }  
  
    public void unregisterObserver(Observer<String> observer) {  
        observers.remove(observer);  
    }  
  
    public void notifyObservers(String event) {               // (2)  
        observers.forEach(observer -> observer.observe(event)); // (2.1)  
    }  
}
```

Как показано в предыдущем примере, реализация `Subject` хранит множество `Set` наблюдателей (1), подписавшихся на получение уведомлений. Наблюдатели могут оформлять или отменять подписку с помощью методов `registerObserver` и `unregisterObserver`. Для рассылки событий в `Subject` предусмотрен метод `notifyObservers` (2), который в цикле обходит список наблюдателей и вызывает метод `observe()` каждого зарегистрировавшегося экземпляра `Observer`, передавая ему актуальное событие (2.1). Для безопасного выполнения в многопоточной среде мы используем `CopyOnWriteArraySet`, потокобезопасную реализацию множества `Set`, которая создает новую копию своих элементов всякий раз, когда выполняется операция `update`. Эта операция обходится довольно дорого, особенно если контейнер `CopyOnWriteArraySet` содержит большое число элементов. Однако обычно список подписчиков редко меняется, поэтому выбор данного контейнера выглядит достаточно разумным для потокобезопасной реализации `Subject`.

Примеры использования шаблона «Наблюдатель»

Теперь напомним простой тест *JUnit*, использующий наши классы и демонстрирующий их взаимодействие. В следующем фрагменте мы также использовали библиотеку Mockito (<http://site.mockito.org>), чтобы подтвердить ожидания.

```
@Test
public void observersHandleEventsFromSubject() {

    // дано
    Subject<String> subject = new ConcreteSubject();
    Observer<String> observerA = Mockito.spy(new ConcreteObserverA());
    Observer<String> observerB = Mockito.spy(new ConcreteObserverB());

    // если
    subject.notifyObservers("No listeners");

    subject.registerObserver(observerA);
    subject.notifyObservers("Message for A");

    subject.registerObserver(observerB);
    subject.notifyObservers("Message for A & B");

    subject.unregisterObserver(observerA);
    subject.notifyObservers("Message for B");

    subject.unregisterObserver(observerB);
    subject.notifyObservers("No listeners");

    // тогда
    Mockito.verify(observerA, times(1)).observe("Message for A");
    Mockito.verify(observerA, times(1)).observe("Message for A & B");
    Mockito.verifyNoMoreInteractions(observerA);

    Mockito.verify(observerB, times(1)).observe("Message for A & B");
    Mockito.verify(observerB, times(1)).observe("Message for B");
    Mockito.verifyNoMoreInteractions(observerB);
}
```

После запуска предыдущий тест производит следующий вывод. Здесь видно, какие сообщения были получены каждым экземпляром *Observer*.

```
Observer A: Message for A
Observer A: Message for A & B
Observer B: Message for A & B
Observer B: Message for B
```

Если в течение всего времени выполнения приложения не потребуется отменять подписку, можно воспользоваться поддержкой лямбда-выражений в Java 8 и заменить реализацию класса `Observer` лямбда-выражениями. Напишем соответствующий тест.

```
@Test
public void subjectLeveragesLambdas() {
    Subject<String> subject = new ConcreteSubject();

    subject.registerObserver(e -> System.out.println("A: " + e));
    subject.registerObserver(e -> System.out.println("B: " + e));
    subject.notifyObservers("This message will receive A & B");
    ...
}
```

Важно отметить, что текущая реализация `Subject` основана на множестве `CopyOnWriteArraySet`, не отличающемся высокой эффективностью. Однако это потокобезопасная реализация, а значит, мы можем использовать `Subject` в многопоточном окружении. Сие пригодится, например, когда события распределяются через множество независимых компонентов, обычно выполняющихся в нескольких потоках (что особенно актуально в наши дни, когда многие приложения являются многопоточными). На протяжении всей книги мы будем поднимать вопросы безопасности в многопоточной среде и освещать другие проблемы, связанные с многопоточностью.

Отметим также, что при наличии большого числа наблюдателей, обрабатывающих события с заметной задержкой, сообщения можно рассылать параллельно, используя отдельный поток или даже пул потоков. В результате может получиться примерно такая реализация метода:

```
private final ExecutorService executorService =
    Executors.newCachedThreadPool();

public void notifyObservers(String event) {
    observers.forEach(observer ->
        executorService.submit(
            () -> observer.observe(event)
        )
    );
}
```

Однако с такими **улучшениями** мы встаем на скользкую дорожку доморощенных решений, которые обычно не отличаются эффективностью и практически наверняка содержат скрытые ошибки. Например, мы можем забыть ограничить размер пула, что в конечном итоге приведет к ошибке `OutOfMemoryError`. `ExecutorService` с простейшими настройками может бесконтрольно создавать новые потоки выполнения, когда клиенты требуют запланировать выполнение

заданий чаще, чем исполнители успевают завершить текущие. А так как каждый поток в Java потребляет около 1 Мбайт памяти, типичное приложение для JVM вполне может исчерпать всю доступную память, создав несколько тысяч потоков.



Более подробную информацию об экспериментах с потоками выполнения в JVM ищите в статье Питера Лори (Peter Lawrey): <http://vanillajava.blogspot.com/2011/07/java-what-is-limit-to-number-of-threads.html>. Она написана довольно давно, но с тех пор модель памяти JVM мало изменилась. Чтобы узнать размер стека по умолчанию в своей конфигурации Java, выполните следующую команду:

```
java -XX:+PrintFlagsFinal -version | grep ThreadStackSize
```

Чтобы предотвратить избыточное использование ресурсов, можно ограничить размер пула потоков и ухудшить **живучесть** приложения. Подобные ситуации возникают, когда все доступные потоки пытаются отправить события одному и тому же медлительному наблюдателю `Observer`. Здесь мы попробовали обозначить лишь некоторые из возможных проблем. Кроме того, как отмечается в статье *«Improved Multithreaded Unit Testing»* (<http://users.ece.utexas.edu/~gligoric/papers/JagannathETAL11IMunit.pdf>), *«многопоточный код очень сложно писать и тестировать»*.

Следовательно, когда возникает потребность в многопоточной реализации шаблона «Наблюдатель», старайтесь использовать проверенные библиотеки.



Определение живучести мы берем из конкурентных вычислений, где она описывается как набор свойств, которыми должна обладать конкурентная система, даже если ее компоненты должны выполнять критические секции. Впервые такое определение дал Лесли Лэмпорт (Lesley Lamport) в статье *«Proving the Correctness of Multiprocess Programs»* (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.9454&rep=rep1&type=pdf>).

Обзор шаблона «Наблюдатель» был бы неполным без упоминания классов `Observer` и `Observable` из пакета `java.util`. Они впервые появились в JDK 1.0, то есть имеют давнюю историю. Если заглянуть в исходный код, то можно увидеть прямолинейную реализацию, очень похожую на ту, о которой говорилось выше. Поскольку эти классы были написаны до появления в Java шаблонных классов (*generics*), они работают с событиями типа `Object` и, следовательно, не обеспечивают безопасность типов. Кроме того, их реализация далеко не самая эффективная, особенно в многопоточном окружении. С учетом вышесказанного данные классы были объявлены устаревшими в Java 9, поэтому нет смысла использовать их в новых приложениях.



Более подробную информацию о том, почему объявлены устаревшими классы `Observer` и `Observable`, можно найти по ссылке <https://dzone.com/articles/javas-observer-and-observable-are-deprecated-in-jd>.

Конечно, разрабатывая приложения, мы можем использовать свои реализации шаблона «Наблюдатель». При таком подходе можно отделить источники событий и наблюдателей. Однако порой очень сложно учесть все аспекты современных многопоточных приложений, такие как обработка ошибок, асинхронное выполнение, потокобезопасность, требование к высокой производительности и т. д. Мы уже видели, что реализации событий в JDK недостаточно для образовательных целей, поэтому предпочтительнее использовать более зрелую реализацию от авторитетного производителя.

Шаблон «Публикация/Подписка» с использованием @EventListener

Слишком утомительно было бы разрабатывать программное обеспечение, если бы каждый раз требовалось снова и снова реализовать одни и те же шаблоны. К счастью, у нас есть Spring Framework – восхитительная коллекция библиотек, а также другие фреймворки (да-да, Spring не единственный). Как мы знаем, Spring Framework предлагает множество строительных блоков, которые могут пригодиться при создании программного обеспечения. И конечно, в фреймворке давно уже появилась своя реализация шаблона «Наблюдатель», которая широко используется в приложениях для поддержки событий жизненного цикла. Начиная с версии Spring Framework 4.2 эта реализация и сопутствующий программный интерфейс (API) были расширены и могут использоваться для обработки не только предопределенных событий внутри приложения, но также событий бизнес-логики. Для поддержки событий в Spring теперь имеется аннотация @EventListener, которой маркируются обработчики событий, и класс ApplicationEventPublisher, реализующий публикацию событий.

Здесь мы должны пояснить, что пара @EventListener и ApplicationEventPublisher реализует шаблон проектирования «Публикация/Подписка» (Publish/Subscribe), который выглядит как разновидность шаблона «Наблюдатель».



Хорошее описание шаблона «Публикация/Подписка» можно найти по ссылке <http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>.

В отличие от шаблона «Наблюдатель», в шаблоне «Публикация/Подписка» издатель и подписчики *ничего не знают друг о друге*. Это демонстрирует рис. 2.3.

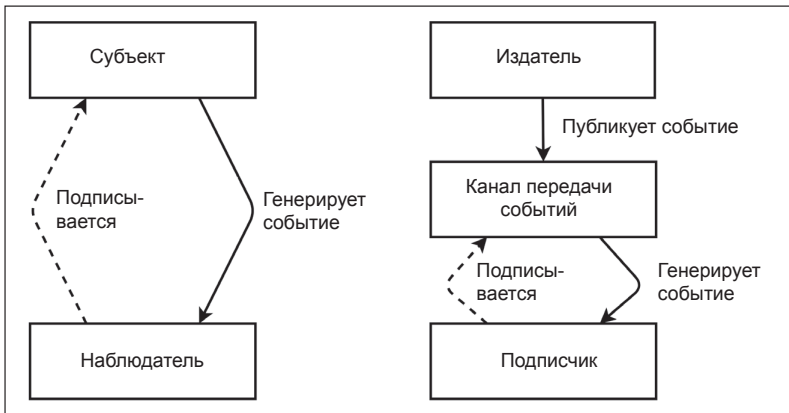


Рис. 2.3. Шаблон «Наблюдатель» (слева) и шаблон «Публикация/Подписка» (справа)

Шаблон «Публикация/Подписка» образует между издателями и подписчиками дополнительный уровень косвенности. Подписчики знают о существовании каналов передачи событий, через которые выполняется рассылка уведомлений, но обычно их не интересует идентичность издателя. Кроме того, к каждому каналу может быть подключено сразу несколько издателей. Диаграмма на рис. 2.3 призвана помочь понять разницу между шаблонами «Наблюдатель» и «Публикация/Подписка». **Канал передачи событий** (также известный как брокер, или шина, событий) дополнительно может фильтровать входящие сообщения и распределять их между подписчиками. Фильтрация и маршрутизация могут осуществляться по содержанию сообщений, по теме или тому и другому сразу. Следовательно, подписчики в такой *тематической* системе будут получать все сообщения, опубликованные под интересующей их темой.

Аннотация `@EventListener` в Spring Framework поддерживает и тематическую, и контентную маршрутизации. Роль тем могут играть типы сообщений, а атрибут `condition` помогает организовать контентную маршрутизацию с использованием **Spring Expression Language (SpEL)**.



В качестве альтернативы реализации шаблона «Публикация/Подписка» из фреймворка Spring можно использовать популярную свободную Java-библиотеку **MBassador**. Ее единственная цель – предложить легковесную и высокопроизводительную шину событий, реализующую шаблон «Публикация/Подписка». Авторы утверждают, что MBassador экономит ресурсы и обладает высокой производительностью, так как библиотека почти не имеет внешних зависимостей и не ограничивает дизайн вашего приложения. За более подробной информацией обращайтесь на страницу проекта в GitHub (<https://github.com/bennidi/mbassador>). Хотелось бы упомянуть и библиотеку Guava, которая тоже предлагает шину событий, реализующую шаблон «Публикация/Подписка». Описание программного интерфейса библиотеки Guava с примерами кода

найдете в следующей статье: <https://github.com/google/guava/wiki/EventBusExplained>.

Создание приложений с @EventListener

Чтобы поэкспериментировать с шаблоном «Публикация/Подписка» в Spring Framework, выполним небольшое упражнение. Предположим, нам нужно реализовать простую веб-службу, возвращающую текущую температуру в помещении. Для измерения температуры у нас имеется температурный датчик, который периодически генерирует события с текущей температурой по шкале Цельсия. Конечно, хотелось бы иметь и мобильное приложение, и веб-приложение, но для краткости реализуем только последнее. Кроме того, поскольку вопросы связи с микроконтроллером выходят за рамки книги, смоделируем датчик температуры с помощью генератора случайных чисел.

Чтобы приложение соответствовало принципам **реактивного дизайна**, мы не можем использовать старую модель извлечения данных для получения информации. К счастью, в настоящее время имеется несколько широко распространенных протоколов для асинхронной передачи сообщений от сервера клиенту, а именно **WebSocket** и **Server-Sent Events (SSE)**. В данном упражнении мы используем последний. Протокол SSE позволяет клиенту получать автоматические обновления с сервера, и он часто используется для передачи в браузер сообщений с обновлениями или непрерывного потока данных. С появлением HTML5 все современные браузеры получили JavaScript API с названием `EventSource`, который используется клиентами для отправки запросов по конкретному адресу URL, чтобы организовать получение потока событий. Кроме того, `EventSource` автоматически восстанавливает соединение, если оно было случайно разорвано из-за каких-то проблем со связью. Важно подчеркнуть, что SSE – отличный кандидат для организации связи между компонентами в реактивной системе. SSE часто будет использоваться в книге наравне с `WebSocket`.



Узнать больше о Server-Sent Events можно в статье «*High Performance Browser Networking*» Ильи Григорика (Ilya Grigorik) по ссылке <https://hpbn.co/server-sent-events-sse/>.

Кроме того, хорошее сравнение `WebSocket` и `Server-Sent Events` найдете в статье Марка Брауна (Mark Brown): <https://www.sitepoint.com/real-time-apps-websockets-server-sent-events/>.

Создание приложения на основе Spring

Для реализации нашего упражнения используем хорошо известные модули Spring: `Spring Web` и `Spring Web MVC`. Наше приложение будет использовать новые возможности Spring 5, поэтому оно будет действовать подобно Spring Framework 4.x. Чтобы упростить процесс разработки, воспользуемся инструментом *Spring*

Boot, который более подробно опишем позже. Чтобы создать приложение, можем настроить и загрузить проект Gradle с веб-сайта Spring Initializer (start.spring.io). После перехода на сайт нужно выбрать желаемую версию Spring Boot и зависимость Web (в конфигурации Gradle эта зависимость получит фактический идентификатор `org.springframework.boot:spring-boot-starter-web`), как показано на рис. 2.4.

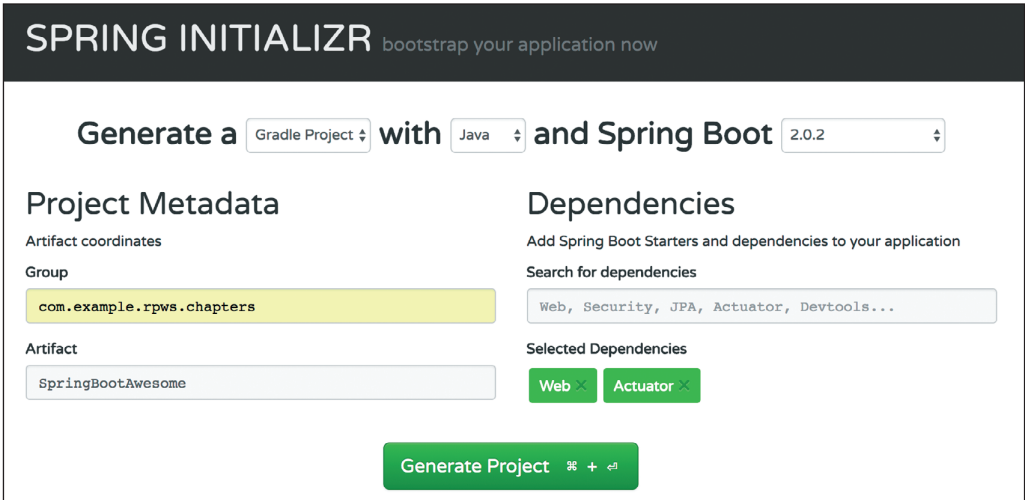


Рис. 2.4. Веб-сайт Spring Initializer упрощает создание нового приложения Spring Boot

Также новый проект Spring Boot можно сгенерировать с помощью cURL и HTTP API сайта Spring Boot Initializer. Следующая команда фактически создаст и загрузит тот же пустой проект со всеми необходимыми зависимостями.

```
curl https://start.spring.io/starter.zip \
  -d dependencies=web,actuator \
  -d type=gradle-project \
  -d bootVersion=2.0.2.RELEASE \
  -d groupId=com.example.rpws.chapters \
  -d artifactId=SpringBootAwesome \
  -o SpringBootAwesome.zip
```

Реализация бизнес-логики

Теперь наметим дизайн нашей системы, как показано на рис. 2.5.

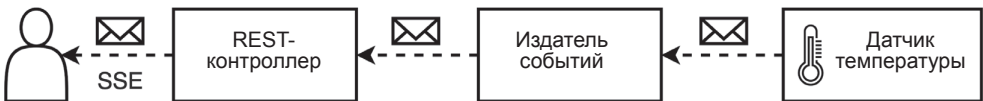


Рис. 2.5. Порядок передачи температуры от датчика к пользователю

В данном случае предметная модель будет состоять из единственного класса `Temperature` с единственным полем типа `double`. Для простоты он также будет использоваться в качестве объекта события, как показано ниже.

```
final class Temperature {  
    private final double value;  
    // конструктор и метод чтения...  
}
```

Для имитации датчика реализуем класс `TemperatureSensor` и снабдим его аннотацией `@Component`, чтобы зарегистрировать компонент Spring.

```
@Component  
public class TemperatureSensor {  
    private final ApplicationEventPublisher publisher;           // (1)  
    private final Random rnd = new Random();                   // (2)  
    private final ScheduledExecutorService executor =           // (3)  
        Executors.newSingleThreadScheduledExecutor();  
  
    public TemperatureSensor(ApplicationEventPublisher publisher) {  
        this.publisher = publisher;  
    }  
  
    @PostConstruct  
    public void startProcessing() {                             // (4)  
        this.executor.schedule(this::probe, 1, SECONDS);  
    }  
  
    private void probe() {                                       // (5)  
        double temperature = 16 + rnd.nextGaussian() * 10;  
        publisher.publishEvent(new Temperature(temperature));  
        // запланировать следующее чтение спустя  
        // случайное число секунд (от 0 до 5)  
  
        executor.schedule(  
            this::probe, rnd.nextInt(5000), MILLISECONDS);    // (5.1)  
        }  
}
```

Итак, наш самодельный датчик температуры зависит только от класса `ApplicationEventPublisher` (1) из Spring Framework. Этот класс дает возможность опубликовать событие в системе. Нам также потребуется генератор случайных чисел (2), чтобы с его помощью генерировать случайные значения температуры из некоторого интервала. Процесс генерации будет выполняться в отдельном экземпляре `ScheduledExecutorService` (3), планирующем создание следующего значения со случайной задержкой (5.1). Вся бизнес-логика сосредоточена в методе `probe()` (5). Упомянутый класс имеет также метод `startProcessing()`, отмеченный аннотацией `@PostConstruct` (4), который вызывается Spring Framework

после подготовки компонента и запускает процесс генерации последовательно-сти случайных значений температуры.

Асинхронные взаимодействия по HTTP с помощью Spring Web MVC

Появившаяся в Servlet 3.0 поддержка асинхронного выполнения добавила возможность обработки HTTP-запросов в потоках выполнения вне контейнера, что очень удобно для выполнения продолжительных операций. Благодаря этим изменениям в Spring Web MVC `@Controller` может возвращать значение не только типа `T`, но и `Callable<T>` и `DeferredResult<T>`. Экземпляр `Callable<T>` может выполняться в потоке за пределами контейнера, но все еще действует в блокирующем стиле. Экземпляр `DeferredResult<T>`, напротив, может генерировать ответы асинхронно в потоке за пределами контейнера, вызывая метод `setResult(T result)`, чтобы его можно было использовать внутри цикла событий.

Начиная с версии 4.2 Spring Web MVC позволяет вернуть `ResponseBodyEmitter`, который действует подобно `DeferredResult`, но может использоваться для отправки нескольких объектов, каждый из которых записывается отдельно с помощью экземпляра преобразователя сообщений (определяется интерфейсом `HttpMessageConverter`).

`SseEmitter` расширяет `ResponseBodyEmitter` и позволяет отправить много сообщений в ответ на один запрос согласно требованиям протокола SSE. Помимо `ResponseBodyEmitter` и `SseEmitter` Spring Web MVC поддерживает также интерфейс `StreamingResponseBody`. При возврате из `@Controller` он позволяет асинхронно отправить необработанные данные. `StreamingResponseBody` может пригодиться для потоковой передачи больших файлов без блокировки потоков сервлетов.

Публикация конечной точки SSE

Далее необходимо добавить класс `TemperatureController` с аннотацией `@RestController`, который будет служить компонентом, обеспечивающим взаимодействия с клиентами по протоколу HTTP.

```
@RestController
public class TemperatureController {
    private final Set<SseEmitter> clients = // (1)
        new CopyOnWriteArraySet<>();

    @RequestMapping(
        value = "/temperature-stream", // (2)
        method = RequestMethod.GET)
    public SseEmitter events(HttpServletRequest request) { // (3)
```

```
SseEmitter emitter = new SseEmitter(); // (4)
clients.add(emitter); // (5)

// Удалить emitter из clients в случае ошибки или разрыва соединения
emitter.onTimeout(() -> clients.remove(emitter)); // (6)
emitter.onCompletion(() -> clients.remove(emitter)); // (7)
return emitter; // (8)
}

@Async // (9)
@EventListener // (10)
public void handleMessage(Temperature temperature) { // (11)
    List<SseEmitter> deadEmitters = new ArrayList<>(); // (12)
    clients.forEach(emitter -> {
        try {
            emitter.send(temperature, MediaType.APPLICATION_JSON); // (13)
        } catch (Exception ignore) {
            deadEmitters.add(emitter); // (14)
        }
    });
    clients.removeAll(deadEmitters); // (15)
}
```

Теперь, чтобы понять логику класса `TemperatureController`, нужно поближе познакомиться с `SseEmitter`. Spring Web MVC предлагает этот класс с единственной целью – дать возможность посылать SSE-события. Когда метод, обрабатывающий запрос, возвращает экземпляр `SseEmitter`, фактическая обработка запроса продолжится, пока не будет вызван метод `SseEmitter.complete()`, не произойдет ошибка или не истечет время ожидания.

`TemperatureController` реализует единственный обработчик запросов (3) для URI `/temperature-stream` (2), который возвращает `SseEmitter` (8). Если клиент пришлет запрос на этот URI, мы создадим и вернем новый экземпляр `SseEmitter` (4), предварительно зарегистрировав его в списке активных клиентов (5). Конструктору `SseEmitter` можно также передать параметр `timeout`.

В качестве коллекции `clients` можно использовать экземпляр класса `CopyOnWriteArraySet` из пакета `java.util.concurrent` (1). Он позволяет одновременно изменять список и выполнять его обход. Когда веб-клиент открывает новый сеанс SSE, мы добавляем новый эмиттер в коллекцию `clients` collection. Удаление экземпляра `SseEmitter` произойдет автоматически по завершении обработки или истечении тайм-аута (6) (7).

Теперь, имея канал связи с клиентами, мы должны организовать получение событий со значениями температуры. Для этого в нашем классе имеется метод `handleMessage()` (11). Он снабжен аннотацией `@EventListener` (10), превра-

шающей его в обработчик событий Spring. Фреймворк будет вызывать метод `handleMessage()` только при получении события `Temperature`, потому что параметр `temperature` метода имеет именно этот тип. Аннотация `@Async` (9) сообщает, что метод может вызываться **асинхронно**, в данном случае он вызывается из пула потоков, сконструированного вручную. Метод `handleMessage()` принимает новое событие со значением температуры и асинхронно рассылает его всем клиентам в формате JSON (13). Также при отправке каждого эмиттера мы определяем неудачные попытки (14) и удаляем эмиттер из списка активных клиентов (15). Такой подход позволяет обнаруживать отключившихся клиентов. К сожалению, `SseEmitter` не поддерживает функций обратного вызова для обработки ошибок, поэтому единственная возможность обработать ошибку в его методе `send()` – перехватить исключение.

Настройка поддержки асинхронного выполнения

Чтобы опробовать все это на практике, нам нужна точка входа в приложение со следующими методами.

```
@EnableAsync // (1)
@SpringBootApplication // (2)
public class Application implements AsyncConfigurer {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public Executor getAsyncExecutor() { // (3)
        ThreadPoolTaskExecutor executor =
            new ThreadPoolTaskExecutor(); // (4)
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(100);
        executor.setQueueCapacity(5); // (5)
        executor.initialize();
        return executor;
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler(){
        return new SimpleAsyncUncaughtExceptionHandler(); // (6)
    }
}
```

Как видите, за основу взято приложение Spring Boot (2) с поддержкой асинхронного выполнения, которая включается аннотацией `@EnableAsync` (1). Здесь можно настроить обработчик исключений, возбуждаемых в процессе асинхронно-

го выполнения (6). Здесь же мы готовим Executor для асинхронной обработки. В данном случае мы использовали ThreadPoolTaskExecutor с двумя основными потоками, число которых может быть увеличено до сотни. Важно отметить, что если неправильно настроить емкость очереди (5), пул потоков не сможет увеличиваться в размерах, так как в этом случае будет использоваться очередь SynchronousQueue, ограничивающая возможность параллельного выполнения.

Создание пользовательского интерфейса с поддержкой SSE

Последнее, что нам осталось сделать, чтобы завершить пример, – создать страницу HTML с некоторым кодом на JavaScript, взаимодействующую с сервером. Для простоты мы убрали все HTML-теги и оставили только самый минимум, необходимый для достижения результата.

```
<body>
  <ul id="events"></ul>
  <script type="application/javascript">
    function add(message) {
      const el = document.createElement("li");
      el.innerHTML = message;
      document.getElementById("events").appendChild(el);
    }

    var eventSource = new EventSource("/temperature-stream"); // (1)

    eventSource.onmessage = e => { // (2)
      const t = JSON.parse(e.data);
      const fixed = Number(t.value).toFixed(2);
      add('Temperature: ' + fixed + ' C');
    }

    eventSource.onopen = e => add('Connection opened'); // (3)
    eventSource.onerror = e => add('Connection closed'); //
  </script>
</body>
```

Здесь мы используем объект EventSource, указывающий на /temperature-stream (1). Он обрабатывает входящие сообщения, вызывая функцию onmessage() (2). Обработка ошибок и реакция на открытие потока данных реализуются похожим образом (3). Мы должны сохранить эту страницу в файле index.html и поместить его в каталог src/main/resources/static/ в папке проекта. По умолчанию Spring Web MVC использует каталог для обслуживания HTTP-запросов. Это поведение можно изменить, предоставив конфигурацию, расширяющую класс WebMvcConfigurerAdapter.

Проверка приложения

После сборки и запуска приложения можем обратиться к нему по адресу: `http://localhost:8080`. (По умолчанию Spring Web MVC использует порт 8080 для веб-сервера, однако его можно изменить в файле `application.properties`, в строке `server.port=8080`.) Спустя несколько секунд можно увидеть вывод приложения.

```
Connection opened
Temperature: 14.71 C
Temperature: 9.67 C
Temperature: 19.02 C
Connection closed
Connection opened
Temperature: 18.01 C
Temperature: 16.17 C
```

Как видите, веб-страница реактивно получает события, экономя ресурсы клиента и сервера. Также поддерживается автоматическое восстановление соединения после тайм-аута, если в сети возникли сбои. Так как текущее решение не зависит напрямую от JavaScript, можем подключиться к серверу с помощью другого клиента, например `curl`. Выполнив следующую команду в терминале, мы получили поток данных с информацией о событиях.

```
> curl http://localhost:8080/temperature-stream
data:{"value":22.33210856124129}
data:{"value":13.83133638119636}
```



Узнать больше о технологии Server-Sent Events и интеграции с Spring Framework можно в замечательной статье Ральфа Шаера (Ralph Schaer): <https://golb.hplar.ch/p/Server-Sent-Events-with-Spring>.

Критический обзор решения

На данном этапе можем поздравить себя с созданием устойчивого реактивного приложения, уместившегося в нескольких десятках строк кода (включая HTML и JavaScript). Однако текущее решение имеет несколько проблем. Прежде всего мы использовали инфраструктуру публикации/подписки из Spring. В Spring Framework этот механизм первоначально предназначался для обработки событий жизненного цикла приложений и не предполагал использования в сценариях, где требуется высокая производительность или способность справляться с высокой нагрузкой. Что случится, если вместо одного потока данных с температурой нам понадобится организовать несколько тысяч или даже миллионов таких потоков? Сможет ли реализация в Spring эффективно справиться с подобной нагрузкой?

Кроме того, у этого подхода есть существенный недостаток – для реализации бизнес-логики мы использовали внутренние механизмы Spring. Таким образом, небольшие изменения во фреймворке могут нарушить работоспособность приложения. Также невозможно организовать модульное тестирование без запуска контекста приложения. Кроме того, как рассказывалось в главе 1 «*Причины выбора Spring*», сложно анализировать код приложения, обладающего большим количеством методов с аннотациями `@EventListener` и не имеющего сценария, четко описывающего рабочий поток в виде компактного фрагмента кода.

Далее `SseEmitter` поддерживает такие понятия, как ошибка и конец потока данных, а `@EventListener` – нет. То есть чтобы сообщить об окончании данных или об ошибке другим компонентам, мы должны определить некоторые специальные объекты или иерархии классов и, которые легко забыть обработать. Кроме того, у таких специфических маркеров немного разная семантика в разных ситуациях, что усложняет решение и снижает привлекательность подхода.

Стоит обратить внимание на еще один недостаток: мы выделяем пул потоков для асинхронной рассылки событий с информацией о температуре. В настоящем асинхронном и реактивном решении нам не потребовалось бы делать это.

Наш датчик температуры генерирует только один поток событий независимо от числа клиентов. Но он генерирует события, даже когда нет ни одного клиента. То есть напрасно расходуются ресурсы, что особенно критично в ресурсоемких задачах. Например, наш компонент может взаимодействовать с реальным оборудованием и сокращать срок его службы.

Для решения этих и других проблем нужна специально разработанная реактивная библиотека. К счастью, у нас на выбор есть несколько таких библиотек. Далее рассмотрим RxJava, первую реактивную библиотеку, получившую широкое распространение и изменившую способ разработки реактивных приложений на Java.

RxJava как реактивный фреймворк

Некоторое время назад появилась стандартная библиотека поддержки реактивного программирования на Java – RxJava 1.x (подробности ищите по ссылке <https://github.com/ReactiveX/RxJava>). Эта библиотека проторила дорогу реактивному программированию в мир Java и придала ему современные черты. В настоящее время это не единственная такая библиотека. У нас также есть Akka Streams и Project Reactor. Последняя подробно описывается в главе 4 «*Project Reactor – основа реактивных приложений*». То есть сейчас можно выбрать из нескольких вариантов. Кроме того, сама RxJava сильно изменилась с выходом версии 2.x. Однако, чтобы понять самые основные идеи реактивного программирования и научиться рассуждать о них, мы сосредоточим все внимание только на самой основополагающей части RxJava – на API, который не изменился с самых

ранних версий библиотеки. Все примеры в этом разделе должны работать с обеими версиями – RxJava 1.x и RxJava 2.x.

Чтобы иметь возможность одновременно существовать в одном пути к классам приложения, у RxJava 2.x и RxJava 1.x разные групповые идентификаторы (`io.reactivex.rxjava2` и `io.reactivex`) и разные пространства имен (`io.reactivex` и `rx`).



Несмотря на то что официальная поддержка RxJava 1.x прекратилась в марте 2018 года, она все еще используется во многих библиотеках и приложениях, в основном потому, что получила очень широкое распространение. Хорошо описывается, что изменилось в RxJava 2.x по сравнению с RxJava 1.x, в следующей статье: <https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0>.

Библиотека RxJava – это реализация реактивных расширений **Reactive Extensions** в Java VM (также известных как **ReactiveX**). Reactive Extensions – набор инструментов, позволяющих императивным языкам работать с потоками данных независимо от того, какая у них природа: синхронная или асинхронная. ReactiveX часто определяется как комбинация шаблонов проектирования «Наблюдатель», «Итератор» и функционального программирования. Дополнительную информацию о ReactiveX найдете на сайте <http://reactivex.io>.

Реактивное программирование может показаться трудным, особенно для тех, кто пришел из императивного мира, но главная его идея относительно проста. Здесь мы познакомимся с основами RxJava – наиболее широко распространенной реактивной библиотеки. Мы не будем рассматривать ее во всех подробностях, а просто перечислим наиболее важные идеи реактивного программирования.

«Наблюдатель» плюс «Итератор» равно «реактивный поток»

В этой главе мы много говорили о шаблоне «Наблюдатель», четко разграничивающем производителя и потребителя события. Вспомним определение интерфейсов, определяемых этим шаблоном.

```
public interface Observer<T> {  
    void notify(T event);  
}  
  
public interface Subject<T> {  
    void registerObserver(Observer<T> observer);  
    void unregisterObserver(Observer<T> observer);  
    void notifyObservers(T event);  
}
```

Как сказано ранее, данный шаблон прекрасно подходит для организации бесконечных потоков данных, но было бы замечательно, если бы имелась возможность сообщить об окончании потока. Также нежелательно, чтобы производитель генерировал события до появления потребителей. В синхронном мире для этого существует шаблон «Итератор». Вот как его можно описать.

```
public interface Iterator<T> {  
    T next();  
    boolean hasNext();  
}
```

Итератор `Iterator` предлагает метод `next()` для получения элементов по одному, а также имеет возможность сообщить об окончании последовательности, вернув значение `false` при вызове метода `hasNext()`. Но что получится, если соединить эту идею с асинхронным выполнением, обеспечиваемым шаблоном «Наблюдатель»? Вот как мог бы выглядеть результат.

```
public interface RxObserver<T> {  
    void onNext(T next);  
    void onComplete();  
}
```

Интерфейс `RxObserver` очень похож на `Iterator`, но теперь клиент не должен вызывать метод `next()` итератора `Iterator`, `RxObserver` сам уведомит его о появлении следующего значения обратным вызовом `onNext()`. Кроме того, клиент не должен проверять результат вызова `hasNext()`, потому что `RxObserver` сообщит об окончании потока данных обратным вызовом `onComplete()`. Все это, конечно, хорошо, но как быть с ошибками? Экземпляр `Iterator` может возбудить исключение в процессе выполнения метода `next()`, поэтому было бы хорошо иметь механизм передачи ошибки от производителя потребителю `RxObserver`. Добавим для этого специальный метод обратного вызова `onError()`. То есть окончательное решение будет таким:

```
public interface RxObserver<T> {  
    void onNext(T next);  
    void onComplete();  
    void onError(Exception e);  
}
```

Так получилось, что мы только что воссоздали интерфейс `Observer`, лежащий в основе `RxJava`. Он определяет порядок передачи данных между частями реактивного потока. Интерфейс `Observer` можно найти повсюду, он очень напоминает интерфейс `Observer` из шаблона «Наблюдатель».

Реактивный класс `Observable` является аналогом `Subject` из шаблона «Наблюдатель». Как следствие `Observable` играет роль источника событий. Он имеет сотни

методов преобразования потока данных и десятки фабричных методов для инициализации реактивного потока.

Абстрактный класс `Subscriber` реализует интерфейс `Observer` и потребляет события. Он также используется как базовый класс для фактической реализации подписчика. Отношения между `Observable` и `Subscriber` регламентируются реализацией подписки `Subscription`, которая позволяет проверять состояние подписки и отменять ее при необходимости. Это отношение показано на рис. 2.6.

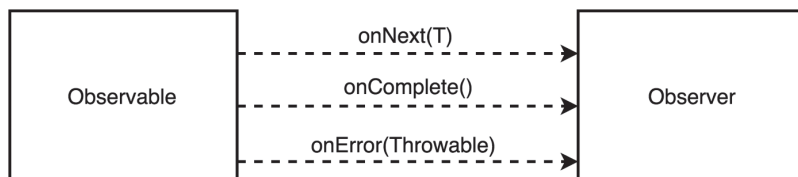


Рис. 2.6. Контракт `Observable/Observer`

RxJava определяет правила, описывающие порядок отправки событий. Источнику `Observable` разрешено отправить любое число (в том числе ноль) событий. Затем он должен сигнализировать об окончании выполнения, послав либо признак успешного завершения, либо ошибку. То есть `Observable` должен вызывать `onNext()` для каждого подключенного абонента любое число раз, а затем `onComplete()` или `onError()` (но не оба). Кроме того, ему запрещено вызывать `onNext()` после `onComplete()` или `onError()`.

Производство и потребление потоков

На данный момент у нас достаточно знаний, чтобы с помощью библиотеки RxJava написать первое простое приложение. Определим поток данных, представленный классом `Observable`. Мы можем предположить, что `Observable` является своеобразным генератором, который знает, как посылать события подписчикам после оформления подписки.

```
Observable<String> observale = Observable.create(  
    new Observable.OnSubscribe<String>() {  
        @Override  
        public void call(Subscriber<? super String> sub) {           // (1)  
            sub.onNext("Hello, reactive world!");                    // (2)  
            sub.onCompleted();                                         // (3)  
        }  
    }  
);
```

Здесь мы создали новый экземпляр `Observable`, определив функцию, которая будет вызываться при появлении каждого подписчика `Subscriber` (1). На данный

момент наш экземпляр `Observable` генерирует одно строковое значение (2) и посылает подписчику сигнал об окончании потока (3). Мы можем усовершенствовать этот код, используя `lambda`-выражения из Java 8.

```
Observable<String> observable = Observable.create(
    sub -> {
        sub.onNext("Hello, reactive world!");
        sub.onCompleted();
    }
);
```

В отличие от Java Stream API экземпляр `Observable` поддерживает возможность повторного использования, и каждый подписчик получит строку `Hello, reactive world!` сразу после подписки.



Обратите внимание, что начиная с версии RxJava 1.2.7 реализация `Observable` считается устаревшей и небезопасной, потому что может сгенерировать слишком много событий, с которыми не справится подписчик. Иначе говоря, этот подход не позволяет оказывать обратное давление, о котором мы подробно расскажем ниже. Однако данный код все еще хорош для знакомства.

Теперь реализуем класс `Subscriber`.

```
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) {                                // (1)
        System.out.println(s);
    }

    @Override
    public void onCompleted() {                                    // (2)
        System.out.println("Done!");
    }

    @Override
    public void onError(Throwable e) {                             // (3)
        System.err.println(e);
    }
};
```

Как видим, `Subscriber` должен реализовать методы интерфейса `Observer` и определить реакцию на новые события (1), сигнал о завершении потока (2) и ошибки (3). Теперь посмотрим, как взаимодействуют экземпляры `observable` и `subscriber`.

```
observable.subscribe(subscriber);
```

Если выполнить этот код, программа выведет следующие строки.

```
Hello, reactive world!  
Done!
```

Ура! Мы только что написали простое реактивное приложение! Как вы уже наверняка поняли, этот код можно переписать с использованием lambda-выражений.

```
Observable.create(  
    sub -> {  
        sub.onNext("Hello, reactive world!");  
        sub.onCompleted();  
    }  
)  
.subscribe(  
    System.out::println,  
    System.err::println,  
    () -> System.out.println("Done!")  
);
```

Библиотека RxJava предлагает большую гибкость в создании экземпляров `Observable` и `Subscriber`. Например, можно создать экземпляр `Observable`, *просто* сославшись на элементы обычного массива или из итерируемой коллекции `Iterable`.

```
Observable.just("1", "2", "3", "4");  
Observable.from(new String[]{"A", "B", "C"});  
Observable.from(Collections.emptyList());
```

Также можно сослаться на экземпляр `Callable` (1) или даже `Future` (2).

```
Observable<String> hello = Observable.fromCallable(() -> "Hello "); // (1)  
Future<String> future =  
    Executors.newCachedThreadPool().submit(() -> "World");  
Observable<String> world = Observable.from(future); // (2)
```

Поток `Observable` можно создать не только простыми способами, но и объединением других экземпляров `Observable` и таким способом воспроизвести довольно сложный рабочий процесс. Например, оператор `concat()` каждого из входящих потоков потребляет все элементы и отправляет их следующему наблюдателю. Входящие потоки будут обрабатываться до тех пор, пока не выполнится завершающая операция (`onComplete()`, `onError()`) в порядке, соответствующем порядку аргументов `concat()`. Ниже приводится пример использования `concat()`.

```
Observable.concat(hello, world, Observable.just("!"))  
    .forEach(System.out::print);
```


Это часть простой комбинации из нескольких экземпляров `Observable`, использующих разные источники. Здесь же мы используем метод `Observable.forEach()` для обхода результатов, как принято в Java 8 Stream API. Данная программа выведет следующее:

Hello World!



Обратите внимание, даже если не определить обработчик ошибок, в случае появления ошибки реализация по умолчанию `Subscriber` возбудит исключение `rx.exceptions.OnErrorNotImplementedException`.

Генерация последовательности асинхронных событий

`RxJava` позволяет генерировать не только одиночные события, но и целые асинхронные последовательности, например через определенные интервалы времени.

```
Observable.interval(1, TimeUnit.SECONDS)
    .subscribe(e -> System.out.println("Received: " + e));
Thread.sleep(5000); // (1)
```

Вот как будет выглядеть вывод этого кода.

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
```

Кроме того, если убрать `Thread.sleep(...)` (1), приложение завершится, так ничего и не выведя. Это объясняется тем, что события генерируются и потребляются отдельным фоновым потоком. Следовательно, чтобы предотвратить преждевременное завершение главного потока, мы должны вызвать `sleep()` или организовать выполнение других полезных действий.

Конечно, существует механизм, управляющий взаимодействием `Observer/Subscriber`. Он называется `Subscription` и имеет следующее определение интерфейса.

```
interface Subscription {
    void unsubscribe();
    boolean isUnsubscribed();
}
```

С помощью метода `unsubscribe()` подписчик `Subscriber` может уведомить источник `Observable`, что больше не нуждается в новых событиях. Иначе гово-

ря, этот метод отменяет подписку. Со своей стороны Observable может вызвать `isUnsubscribed()`, чтобы убедиться, что подписчик Subscriber все еще ожидает новых событий.

Чтобы воочию увидеть, как осуществляется отмена подписки, рассмотрим ситуацию, когда подписчик интересуется лишь частью событий и потребляет их, пока не получит внешний сигнал `CountDownLatch` (1). Входящий поток генерирует новые события каждые 100 мс и образует бесконечную последовательность – 0, 1, 2, 3... (3). Следующий код демонстрирует, как оформить подписку (2) при определении реактивного потока. Он также показывает, как отписаться от потока событий (4).

```
CountDownLatch externalSignal = ...;                                // (1)

Subscription subscription = Observable                             // (2)
    .interval(100, MILLISECONDS)                                // (3)
    .subscribe(System.out::println);

externalSignal.await();
subscription.unsubscribe();                                        // (4)
```

Итак, здесь подписчик получает события 0, 1, 2, 3 и затем сигнал `externalSignal`, что приводит к отмене подписки.

Теперь мы знаем, что реактивная система включает поток Observable, подписчика Subscriber и некоторый экземпляр подписки Subscription, которая сообщает производителю Observable о намерении подписчика Subscriber получать события. Настало время познакомиться с приемами преобразования данных в реактивном потоке.

Преобразование потоков и диаграммы Marble

Классы Observable и Subscriber позволяют реализовать рабочие процессы с самой разной конфигурацией, но истинная мощь библиотеки RxJava скрыта в ее операторах. Операторы используются для преобразования отдельных элементов или изменения структуры потока. В RxJava имеется огромное количество операторов практически на все случаи жизни, однако их исследование выходит за рамки книги. Познакомимся лишь с самыми основными операторами (большинство других – это комбинация данных операторов).

Оператор map

Вне всяких сомнений, чаще других операторов из библиотеки RxJava используется оператор `map` со следующей сигнатурой.

```
<R> Observable<R> map(Func1<T, R> func)
```

Это объявление означает, что функция `func` может преобразовывать объекты типа `T` в объекты типа `R`, а сам метод применяет преобразование `Observable<T>` в `Observable<R>`. Однако сигнатура не всегда хорошо описывает поведение оператора, особенно если последний выполняет сложное преобразование. С этой целью были изобретены **диаграммы Marble**. Они визуально описывают, как происходит преобразование потока. Диаграммы Marble настолько эффективно описывают поведение оператора, что были добавлены в документацию Javadoc всех операторов RxJava. Для оператора `map`, в частности, приводится диаграмма, изображенная на рис. 2.7.

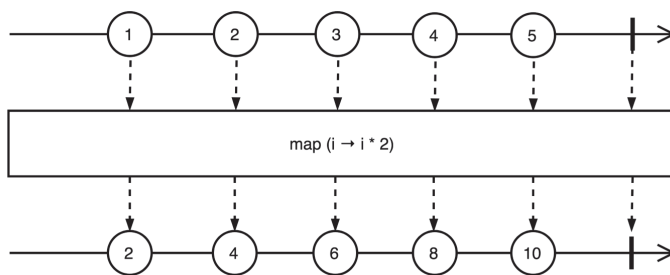


Рис. 2.7. Оператор `map`: преобразует элементы потока, отправляемые экземпляром `Observable`, применяя к каждому указанную функцию

Глядя на рис. 2.7, можно уверенно сказать, что `map` выполняет преобразование «один к одному». Кроме того, выходной поток имеет то же количество элементов, что и входной.

Оператор `filter`

В отличие от `map`, оператор `filter` может произвести меньше элементов, чем получено им. Он отправляет только те элементы, которые преодолели проверку с использованием предиката, как показано на рис. 2.8.

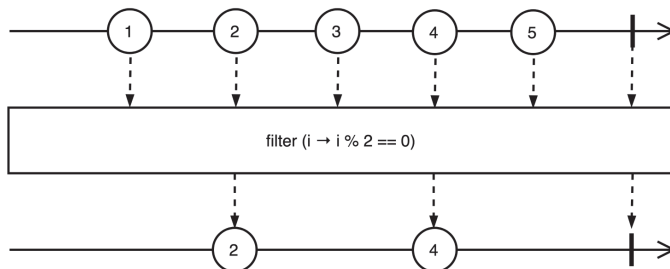


Рис. 2.8. Оператор `filter`: отправляет только те элементы, которые преодолели проверку с использованием предиката

Оператор count

Имя оператора count говорит само за себя: он посылает значение с числом элементов во входном потоке. Но имейте в виду, что отправка числа оператором count произойдет в тот момент, когда исходный поток закончится. То есть для бесконечного потока оператор count никогда ничего не пошлет. Это показано на рис. 2.9.

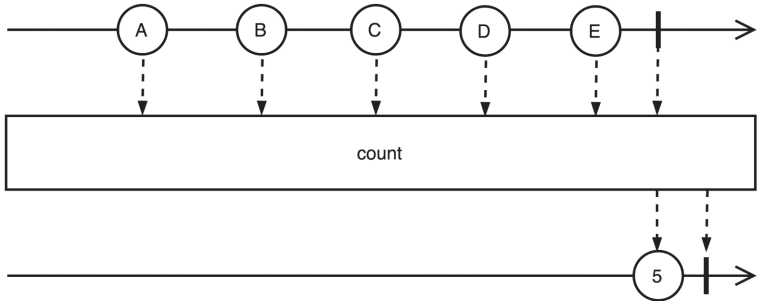


Рис. 2.9. Оператор count: подсчитывает число элементов, отправленных источником Observable, и сам посылает только одно значение

Оператор zip

Рассмотрим еще один оператор – zip. Он обладает сложным поведением: объединяет элементы из двух параллельных потоков, применяя функцию zip. Этот оператор часто используется для обогащения данных, особенно когда разные части результата извлекаются из разных источников, как показано на рис. 2.10.

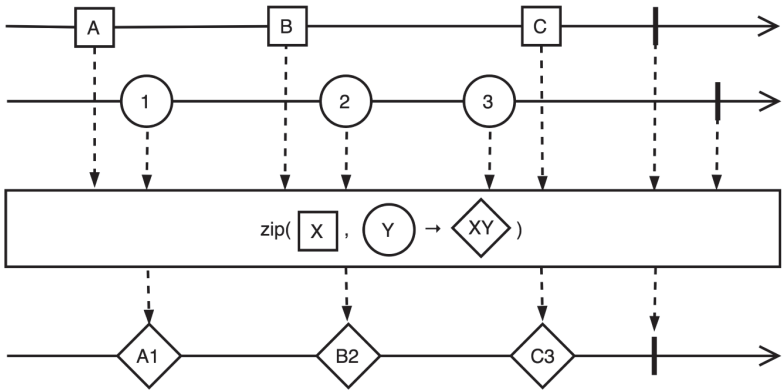


Рис. 2.10. Оператор zip: объединяет несколько потоков Observables с помощью указанной функции и для каждой комбинации посылает единственный элемент – результат этой функции

Netflix, например, использует оператор `zip` для объединения описаний фильмов, киноафиш и рейтингов, чтобы вернуть поток рекомендуемых фильмов. Мы же ради простоты объединим только два потока строковых значений.

```
Observable.zip(
    Observable.just("A", "B", "C"),
    Observable.just("1", "2", "3"),
    (x, y) -> x + y
).forEach(System.out::println);
```

Этот код объединит элементы из двух потоков, как показано на рис. 2.10, и выведет следующие строки:

```
A1
B2
C3
```

Узнать больше операторов, которые часто используются в реактивном программировании (не только в RxJava), можно на сайте <http://rxmarbles.com>. Там имеются интерактивные диаграммы, отражающие фактическое поведение операторов. Кроме того, интерактивный пользовательский интерфейс позволяет визуально показать, как протекает преобразование событий с учетом порядка их появления в потоках. Обратите внимание, что сам сайт построен с использованием библиотеки RxJS (см. <https://github.com/ReactiveX/rxjs>) – аналога RxJava в мире JavaScript.

Как уже отмечалось, класс `Observable` из RxJava предлагает несколько десятков операторов преобразования потоков, которые способны покрыть все основные потребности. Конечно, RxJava не ограничивает разработчиков только теми операторами, которые она сама предоставляет. Вы можете писать свои операторы, реализуя класс, производный от `Observable.Transformer<T, R>`. Логика такого оператора можно включить в рабочий процесс, применив оператор `Observable.compose(transformer)`. Однако прямо сейчас мы не будем углубляться в теорию или практику разработки операторов, вернемся к этой теме в следующих главах. А пока просто подчеркнем, что RxJava предлагает набор надежных инструментов для создания сложных асинхронных процессов, ограниченных лишь вашим воображением, но не библиотекой.

Требования и преимущества RxJava

На примере библиотеки RxJava мы познакомились с основами *реактивного программирования*. Разные реактивные библиотеки могут иметь немного разные API и обладать каждая своими особенностями, но сама идея остается неизменной: подписчик подписывается на получение событий из потока, который, в свою очередь, запускает процесс генерации асинхронных событий. Производитель и потребитель обычно оформляют некую подписку, которая позволяет разорвать от-

ношения «издатель-подписчик». Это очень гибкий подход, он позволяет контролировать количество производимых и потребляемых событий и экономить такты процессора, которые обычно впустую расходуются на создание никому не нужных данных.

Чтобы убедиться, что реактивное программирование позволяет экономить вычислительные ресурсы, представим, что мы должны реализовать простую службу поиска. Она будет возвращать коллекцию адресов URL документов, содержащих искомую фразу. Обычно клиентское приложение дополнительно устанавливает некоторое ограничение, например максимальное число полезных результатов. Без применения методов реактивного программирования мы могли бы реализовать такую службу со следующим API.

```
public interface SearchEngine {  
    List<URL> search(String query, int limit);  
}
```

Как можно заметить из определения интерфейса, служба выполняет поиск, собирает все результаты с учетом установленного ограничения, помещает их в список `List` и возвращает клиенту. В этом сценарии клиент получит полный список с результатами, даже при том что пользователя заинтересует только первый или второй результат на странице. То есть служба проделала большой объем работы, пользователь был вынужден довольно долго ждать результатов, а в конечном счете большая их часть осталась невостребованной. Это, без сомнения, пустая трата ресурсов.

Однако можно пойти другим путем и обрабатывать результаты поиска итеративно. При таком подходе сервер будет искать следующий результат, пока клиент потребляет предыдущий. Обычно поиск на сервере возвращает не один, а целую группу результатов (например, 100). Такое решение называется **курсором** и часто используется в базах данных. С точки зрения клиента курсор – это **итератор**. Ниже приводится версия API нашей службы, улучшенная с учетом вышесказанного.

```
public interface IterableSearchEngine {  
    Iterable<URL> search(String query, int limit);  
}
```

Единственный недостаток решения на основе итератора – блокировка клиентского потока выполнения в режиме активного ожидания, когда придет время получить новую порцию данных. Это было бы катастрофой для пользовательского интерфейса Android. После отправки новой порции результатов служба поиска переходит в ожидание следующего вызова `next()`. Иначе говоря, посредством интерфейса `Iterable` клиент и служба играют в пинг-понг. Иногда такой порядок взаимодействий вполне приемлем, но чаще он оказывается недостаточно эффективным для высокопроизводительных приложений.

В свою очередь, служба поиска может вернуть `CompletableFuture` и таким образом стать асинхронной. В таком случае поток выполнения клиента мог бы занять-

ся чем-нибудь полезным и не ждать результатов поиска, потому что служба автоматически выполнит обратный вызов, когда результаты будут получены. Но здесь мы снова получаем все или ничего, потому что `CompletableFuture` может хранить только одно значение, даже если это список результатов.

```
public interface FutureSearchEngine {  
    CompletableFuture<List<URL>> search(String query, int limit);  
}
```

Задействовав RxJava, можно улучшить данное решение и организовать асинхронную обработку, получив возможность реагировать на каждое прибывающее событие. Кроме того, клиент может отменить подписку вызовом `unsubscribe()` в любой момент и сократить объем работы, выполняемой службой поиска.

```
public interface RxSearchEngine {  
    Observable<URL> search(String query);  
}
```

При таком подходе мы значительно улучшаем отзывчивость приложения. Даже если клиент не получил всех результатов, он имеет возможность обрабатывать имеющиеся. Люди не любят долго ждать и оценивают быстроедействие по таким критериям, как *время до получения первого байта* или *критический путь до отображения* (Critical Rendering Path, CRP). Во всех этих случаях реактивное программирование показывает по крайней мере не худший результат по сравнению с традиционными подходами, а часто и превосходит их.



Более подробную информацию об оценке *времени до получения первого байта* можно найти по ссылке <https://www.maxcdn.com/one/visual-glossary/time-to-first-byte>. Информацию об оценке критического пути до отображения можно найти по ссылке <https://developers.google.com/web/fundamentals/performance/critical-rendering-path>.

Как мы уже видели, RxJava позволяет гибко компоновать потоки данных множеством самых разных способов. Точно так же можно завернуть код, написанный в традиционном синхронном стиле, в асинхронный рабочий процесс. Для управления фактическим потоком выполнения с медлительным `Callable` можно использовать оператор `subscriberOn(Scheduler)`. Он определяет, какой планировщик `Scheduler` (реактивный аналог `ExecutorService` в Java) управляет обработкой потока данных. Более подробно о планировании потоков выполнения поговорим в главе 4 «*Project Reactor – основа реактивных приложений*». Следующий код демонстрирует такую ситуацию.

```
String query = ...;  
Observable.fromCallable(() -> doSlowSyncRequest(query))  
    .subscribeOn(Schedulers.io())  
    .subscribe(this::processResult);
```

Конечно, при подобном подходе нельзя быть уверенным, что весь запрос обрабатывает один поток выполнения. Рабочий процесс может начинаться в одном потоке выполнения, пересекать несколько других потоков и завершаться в совершенно другом, вновь запущенном потоке выполнения. То есть при таком подходе довольно опасно изменять объекты, и единственной разумной стратегией является соблюдение *неизменчивости*. Это не новая идея, а один из основных принципов *функционального программирования*. После создания объект не должен изменяться. Это простое правило помогает избавиться от целого класса проблем, характерных для параллельных приложений.

До появления поддержки *lambda-выражений* в Java 8 было трудно использовать всю мощь реактивного и функционального программирования. Без *lambda-выражений* нам пришлось бы написать множество анонимных или вложенных классов, захламляющих код приложения и содержащих больше шаблонного кода, чем необходимо. На момент создания RxJava проект Netflix широко использовал Groovy для разработки, несмотря на его медлительность, в основном потому, что этот язык поддерживал *lambda-выражения*. Это красноречиво говорит нам, что для успешного использования приемов реактивного программирования очень желательно иметь в языке поддержку представления функций как значений. К счастью, это перестало быть проблемой для Java даже на платформе Android, где существуют такие проекты, как Retrolambda (<https://github.com/orfjackal/retrolambda>), реализующие поддержку *lambda-выражений* для старых версий Java.

Переделка приложения с RxJava

Чтобы получить более полное представление о библиотеке RxJava, перепишем предыдущее приложение измерения температуры с использованием RxJava. Чтобы задействовать библиотеку, нужно добавить в файл `build.gradle` следующую зависимость.

```
compile('io.reactivex:rxjava:1.3.8')
```

В новом приложении используем тот же класс для представления текущей температуры.

```
final class Temperature {  
    private final double value;  
    // конструктор и метод чтения...  
}
```

Реализация бизнес-логики

Класс `TemperatureSensor` прежде отправлял события в `ApplicationEventPublisher`, а теперь он должен возвращать реактивный поток с событиями

Temperature. Реактивная реализация TemperatureSensor могла бы выглядеть примерно так.

```
@Component // (1)
public class TemperatureSensor {
    private final Random rnd = new Random(); // (2)

    private final Observable<Temperature> dataStream = // (3)
        Observable
            .range(0, Integer.MAX_VALUE) // (4)
            .concatMap(tick -> Observable // (5)
                .just(tick) // (6)
                .delay(rnd.nextInt(5000), MILLISECONDS) // (7)
                .map(tickValue -> this.probe())) // (8)
            .publish() // (9)
            .refCount(); // (10)

    private Temperature probe() {
        return new Temperature(16 + rnd.nextGaussian() * 10); // (11)
    }

    public Observable<Temperature> temperatureStream() { // (12)
        return dataStream;
    }
}
```

Здесь мы регистрируем TemperatureSensor как компонент Spring, добавив аннотацию @Component (1), чтобы автоматически связать его с другими компонентами. Реализация TemperatureSensor использует RxJava API, подробно описанный выше. Тем не менее мы поясним используемое преобразование в процессе исследования логики класса.

Наш датчик содержит генератор случайных чисел rnd для имитации физических измерений температуры (2). В инструкции (3) объявляется приватное поле с именем dataStream, значение которого возвращается общедоступным методом temperatureStream() (12). То есть dataStream – единственный поток Observable, определяемый компонентом. Он генерирует фактически бесконечную последовательность чисел (4), применяя фабричный метод range(0, Integer.MAX_VALUE). Метод range() генерирует последовательность целых чисел начиная с 0, которая содержит Integer.MAX_VALUE элементов. К каждому числу применяется преобразование (5) – concatMap(tick -> ...). Метод concatMap() принимает функцию f и преобразует элемент tick в элементы потока Observable, применяя функцию f к каждому элементу входного потока и объединяя их друг за другом в выходной поток. В нашем случае функция f генерирует замеры температуры после случайной задержки (чтобы повторить поведение предыдущей реализации). Чтобы задействовать датчик, создается новый поток с единственным элементом tick (6). Для имитации случайной задержки использована функция delay(rnd.nextInt(5000), MILLISECONDS) (7).

Далее мы проверяем датчик и извлекаем значение температуры, применяя преобразование `map(tickValue -> this.probe())` (8), которое, в свою очередь, вызывает метод `probe()` с прежней логикой генерации температуры (11). В данном случае игнорируем `tickValue`, так как это значение необходимо только для создания потока с единственным элементом. Итак, после вызова `concatMap(tick -> ...)` получаем поток данных, который возвращает значения из датчика через случайные промежутки от одной до пяти секунд.

На самом деле можно было бы вернуть поток без применения операторов (9) и (10), но в данном случае каждый подписчик (клиент SSE) мог бы инициировать новую подписку и создать новую последовательность операций чтения датчика. То есть замеры температуры не будут использоваться совместно всеми подписчиками, что может вызвать перегрузку оборудования и ухудшение его работы. Для предотвращения этой неприятности используем оператор `publish()` (9), который рассылает события из одного исходного потока во все потоки, связанные с подписчиками. Оператор `publish()` возвращает особую разновидность `Observable` – `ConnectableObservable`. Она поддерживает оператор `refCount()` (10), который создает подписку на общий входящий поток, только когда имеется хотя бы одна исходящая подписка. В отличие от типичной реализации шаблона «Издатель/Подписчик», этот прием позволяет не обращаться к датчику в отсутствие клиентов.

Нестандартный SseEmitter

Имея `TemperatureSensor`, представляющий поток значений температуры, можем подписать каждого нового получателя `SseEmitter` на поток данных `Observable` и посылать сигналы `onNext` клиентам SSE. Для обработки ошибок и правильного закрытия HTTP-соединения напомним следующее расширение `SseEmitter`.

```
class RxSeeEmitter extends SseEmitter {
    static final long SSE_SESSION_TIMEOUT = 30 * 60 * 1000L;
    private final Subscriber<Temperature> subscriber;           // (1)

    RxSeeEmitter() {
        super(SSE_SESSION_TIMEOUT);                             // (2)

        this.subscriber = new Subscriber<Temperature>() {       // (3)
            @Override
            public void onNext(Temperature temperature) {
                try {
                    RxSeeEmitter.this.send(temperature);        // (4)
                } catch (IOException e) {
                    unsubscribe();                               // (5)
                }
            }
        }
    }
}
```

```
@Override
public void onError(Throwable e) { } // (6)

@Override
public void onCompleted() { } // (7)
};

onCompletion(subscriber::unsubscribe); // (8)
onTimeout(subscriber::unsubscribe); // (9)
}

Subscriber<Temperature> getSubscriber() { // (10)
    return subscriber;
}
}
```

Класс `RxSeeEmitter` расширяет известный класс `SseEmitter`. Он также инкапсулирует реализацию подписчика на события `Temperature` (1). Конструктор `RxSeeEmitter` вызывает конструктор суперкласса и устанавливает тайм-аут сеанса SSE (2), а также создает экземпляр класса `Subscriber<Temperature>` (3). Этот экземпляр подписчика реагирует на сигналы `onNext`, пересылая их клиенту SSE (4). Если отправка данных не удалась, подписчик отменяет подписку на входящий поток `Observable` (5). В текущей реализации известно, что поток с температурой бесконечен и не может производить никаких ошибок, поэтому обработчики `onComplete()` и `onError()` оставлены пустыми (6), (7), но в действующих приложениях желательно предусмотреть некоторую обработку.

Строки (8) и (9) регистрируют действия, которые требуется выполнить по завершении сеанса SSE, когда закончатся данные или обнаружится тайм-аут. Подписчики `RxSeeEmitter` должны отменять подписку. Для получения ссылки на подписчика `RxSeeEmitter` предлагает метод `getSubscriber()` (10).

Публикация конечной точки SSE

Чтобы опубликовать конечную точку SSE, нужно создать REST-контроллер, который автоматически будет связан с экземпляром `TemperatureSensor`. Далее показано определение такого контроллера, использующего `RxSeeEmitter`.

```
@RestController
public class TemperatureController {

    private final TemperatureSensor temperatureSensor; // (1)

    public TemperatureController(TemperatureSensor temperatureSensor) {
        this.temperatureSensor = temperatureSensor;
    }

    @RequestMapping(
```

```
        value = "/temperature-stream",
        method = RequestMethod.GET) {
    public SseEmitter events(HttpServletRequest request) {
        RxSseEmitter emitter = new RxSseEmitter();           // (2)

        temperatureSensor.temperatureStream()                // (3)
            .subscribe(emitter.getSubscriber());             // (4)

        return emitter;                                       // (5)
    }
}
```

Класс `TemperatureController` – это тот же самый контроллер Spring Web MVC, что и в предыдущем примере. Он хранит ссылку на компонент `TemperatureSensor` (1). Когда создается новый сеанс SSE, контроллер создает экземпляр `RxSseEmitter` (2) и подписывает подписчика в `RxSseEmitter` (4) на поток значений температуры из экземпляра `TemperatureSensor` (3). Затем возвращает экземпляр `RxSseEmitter` контейнеру сервлета для обработки (5).

Как видите, при использовании библиотеки `RxJava` REST-контроллер получается проще, он не должен заботиться об уничтожении ненужных экземпляров `SseEmitter` и синхронизации. Реактивная реализация сама управляет получением значений `TemperatureSensor` и их публикацией. Экземпляр `RxSseEmitter` преобразует реактивные потоки в исходящие сообщения SSE, а `TemperatureController` просто связывает новый сеанс SSE с новым `RxSseEmitter`, подписанным на поток значений температуры. Кроме того, эта реализация не использует шину событий из Spring, поэтому она более переносима и может тестироваться без инициализации контекста Spring.

Конфигурация приложения

Здесь мы не используем подход, основанный на публикации/подписке, и аннотацию `@EventListener`, поэтому не зависим от поддержки асинхронного выполнения. Соответственно, конфигурация приложения стала проще.

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Как видите, на этот раз нет необходимости включать поддержку асинхронного выполнения с помощью аннотации `@EnableAsync`, а также настраивать `Executor` из Spring для обработки событий. Конечно, при необходимости можно настроить `Scheduler` из `RxJava` для управления потоками выполнения, обрабатывающими реактивные потоки данных, но эти настройки не зависят от Spring Framework.

Нам также не придется менять код, реализующий пользовательский интерфейс приложения, он должен работать, как и прежде. Здесь мы должны подчеркнуть, что реализация на основе RxJava не обращается к датчику температуры в отсутствие подписчиков. Такое поведение является естественным следствием наличия в *реактивном программировании* такого понятия, как *активная подписка*. Реализации на основе шаблона «Публикация/Подписка» не обладают таким свойством и потому более ограничены.

Краткая история развития реактивных библиотек

Теперь, когда мы познакомились с RxJava и даже написали несколько реактивных приложений, посмотрим, как рождалось реактивное программирование и для решения каких проблем предназначалось.

Любопытно, что история RxJava и современного реактивного программирования началась в недрах Microsoft. В 2005 году Эрик Мейер (Erik Meijer) и его отдел облачного программирования экспериментировали с моделями программирования для создания *крупномасштабных асинхронных интернет-архитектур, предназначенных для обработки больших объемов данных*. Эксперименты продолжались несколько лет, и вот летом 2007 года на свет появилась первая версия библиотеки Rx. Следующие два года были посвящены проработке разных аспектов библиотеки, включая многопоточность и кооперативную многозадачность. Первую публичную версию *Rx.NET* выпустили в свет 18 ноября 2009 года. Позднее в Microsoft перенесли библиотеку на разные языки программирования, такие как JavaScript, C++, Ruby и Objective-C, а также на платформу *Windows Phone*. Когда Rx приобрела достаточно высокую популярность, Microsoft открыла исходный код Rx.NET осенью 2012 года.



Более подробно о рождении библиотеки Rx рассказывает Эрик Мейер (Erik Meijer) в предисловии к книге Томаша Нуркевича (Tomasz Nurkiewicz) и Бена Кристенсена (Ben Christensen) «*Reactive Programming with RxJava*»².

В какой-то момент идеи Rx просочились за пределы Microsoft, и в 2012 году Пол Беттс (Paul Betts) с Джастином Спаром-Саммерсом (Justin Spahr-Summers) из компании GitHub Inc. написали и выпустили библиотеку ReactiveCocoa для Objective-C. В то же время Бен Кристенсен (Ben Christensen) из Netflix перенес Rx.NET на платформу Java и в начале 2013 года опубликовал исходный код RxJava на GitHub.

Примерно в то же время Netflix столкнулся с очень сложной проблемой обработки огромного объема интернет-трафика, порождаемого потоковыми медиа. Асин-

² Нуркевич Т., Кристенсен Б. Реактивное программирование с использованием RxJava. М.: ДМК Пресс, 2017. – Прим. перев.

хронная реактивная библиотека RxJava помогла им построить реактивную систему, которая в 2015 году обслуживала 37% всего интернет-трафика в Северной Америке! В настоящее время значительная доля трафика в системе обрабатывается с помощью RxJava. Чтобы выдержать эту огромную нагрузку, разработчикам в Netflix пришлось придумать новые архитектурные шаблоны и реализовать их в виде библиотек. Наиболее широко известны следующие библиотеки:

- **Hystrix**: библиотека для реализации отказоустойчивых служб (<https://github.com/Netflix/Hystrix>);
- **Ribbon**: библиотека RPC с поддержкой балансировки нагрузки (<https://github.com/Netflix/ribbon>);
- **Zuul**: защищенный и устойчивый шлюз, поддерживающий динамическую маршрутизацию и возможность мониторинга (<https://github.com/Netflix/zuul>);
- **RxNetty**: реактивный адаптер для *Netty*, клиент/серверного фреймворка NIO (<https://github.com/ReactiveX/RxNetty>).

RxJava является ключевым ингредиентом всех этих библиотек и, как следствие, всей экосистемы Netflix. Успех Netflix в развитии микросервисов и потоковых архитектур подтолкнул другие компании к внедрению таких же подходов, включая RxJava.

На сегодняшний день RxJava используется в некоторых драйверах NoSQL для Java, таких как *Couchbase* (<https://blog.couchbase.com/why-couchbase-chose-rxjava-new-java-sdk/>) и *MongoDB* (<https://mongodb.github.io/mongo-java-driver-rx/>).

Также важно отметить, что RxJava была с воодушевлением встречена разработчиками для Android и такими компаниями, как SoundCloud, Square, NYT и SeatGeek, занимающимися созданием мобильных приложений с RxJava. Это привело к созданию библиотеки под названием *RxAndroid*, значительно упростившей процесс разработки реактивных приложений для Android. Разработчики для iOS получили библиотеку RxSwift – реактивную версию библиотеки Swift.

В настоящее время трудно найти популярный язык программирования, на который не была бы перенесена библиотека Rx. В мире Java, например, имеются библиотеки RxScala, RxGroovy, RxClojure, RxKotlin и RxJRuby, и это далеко не полный список. Чтобы найти реализацию Rx для своего языка, обратитесь сюда: <http://reactivex.io/languages.html>.

Неверно думать, что RxJava была первой и единственной библиотекой реактивного программирования. Важно отметить, что широкое распространение асинхронного программирования помогло создать прочную основу для разработки реактивных методов. Самый значимый, пожалуй, вклад в этом направлении внесла библиотека **NodeJS** и ее сообщество (<https://nodejs.org>).

Реактивный ландшафт

В предыдущих разделах мы узнали, как использовать библиотеку RxJava и объединить ее с Spring Web MVC. Для демонстрации преимуществ мы изменили наше приложение слежения за температурой и усовершенствовали его дизайн, применив RxJava. Однако стоит отметить, что Spring Framework и RxJava не единственная возможная комбинация. Многие серверы приложений тоже используют реактивные подходы. Примером может служить успешный реактивный сервер *Ratpack*, также использующий RxJava.

Наряду с обратными вызовами и API на основе объектов Promise проект Ratpack предлагает RxRatpack – отдельный модуль, позволяющий легко преобразовать Promise из Ratpack в Observable из RxJava и наоборот, как показано ниже.

```
Promise<String> promise = get(() -> "hello world");
RxRatpack
    .observe(promise)
    .map(String::toUpperCase)
    .subscribe(context::render);
```



Узнать больше о сервере Ratpack можно на официальном сайте: <https://ratpack.io/manual/current/all.html>.

Другим примером является известный в мире Android HTTP-клиент Retrofit, который включает обертку RxJava вокруг собственной реализации Future и Callback. Следующий фрагмент показывает, что в Retrofit можно использовать по меньшей мере четыре разных стиля программирования.

```
interface MyService {
    @GET("/user")
    Observable<User> getUserWithRx();

    @GET("/user")
    CompletableFuture<User> getUserWithJava8();

    @GET("/user")
    ListenableFuture<User> getUserWithGuava();

    @GET("/user")
    Call<User> getUserNatively()
}
```

Даже притом что RxJava может улучшить любое решение, реактивный ландшафт не ограничивается только этой библиотекой или ее обертками. В мире JVM есть много других библиотек и серверов, определяющих свои реактивные реализации. На-

пример, хорошо известный реактивный сервер Vert.x очень недолго использовал взаимодействия на основе обратных вызовов, а потом создал свое решение в виде пакета `io.vertx.core.streams`, поддерживающего следующие интерфейсы:

- `ReadStream<T>`: представляет поток элементов, доступных для чтения;
- `WriteStream<T>`: представляет поток, куда можно записывать элементы данных;
- `Pump`: используется для перемещения данных из `ReadStream` в `WriteStream` и управления потоком.

Рассмотрим фрагмент, демонстрирующий использование Vert.x:

```
public void vertexExample(HttpClientRequest request, AsyncFile file) {  
    request.setChunked(true);  
    Pump pump = Pump.pump(file, request);  
    file.endHandler(v -> request.end());  
    pump.start();  
}
```



Eclipse Vert.x – фреймворк для создания приложений, управляемых событиями, дизайном напоминающий Node.js. Он предлагает простую модель конкурентного выполнения, примитивы асинхронного программирования и шину распределения событий, доступную из JavaScript в браузере. Узнать больше о сервере Vert.x и об особенностях реализации реактивных потоков в нем вы сможете здесь: <http://vertx.io/docs/>.

Число альтернативных реализаций библиотеки RxJava и проектов, использующих ее, огромно и не ограничивается перечисленными решениями. Многие компании и проекты с открытым исходным кодом создали свои решения, похожие на RxJava, или дополнили упомянутые выше.

Безусловно, в естественной эволюции и конкуренции библиотек нет ничего плохого, но вы неизбежно столкнетесь с проблемами, попытавшись совместить несколько разных реактивных библиотек или фреймворков в одном приложении на Java. Более того, в конечном счете вы обнаружите, что все реактивные библиотеки имеют схожее поведение, оно отличается лишь деталями. Такая ситуация может поставить под угрозу весь проект, потому что скрытые ошибки будет трудно обнаружить и устранить. То есть независимо от схожести API не стоит смешивать несколько реактивных библиотек (например, Vert.x и RxJava) в одном проекте. К настоящему времени назрела необходимость организовать единый стандартный API для всего реактивного ландшафта, который обеспечит совместимость любых реализаций. Конечно, такой стандарт был разработан – он получил название **Reactive Streams**. В следующей главе рассмотрим его во всех деталях.

Заключение

В этой главе мы рассмотрели несколько известных шаблонов проектирования от «Банды четырех», в том числе «Наблюдатель», «Публикация/Подписка» и «Итератор», лежащих в основе реактивного программирования. Чтобы познакомиться с сильными и слабыми сторонами уже имеющихся инструментов асинхронного программирования, мы написали несколько примеров. Также использовали поддержку ServerSent Events и WebSocket в Spring Framework и поэкспериментировали с шиной событий. Кроме того, мы воспользовались инструментом Spring Boot и `start.spring.io` для быстрого создания заготовки приложения. Несмотря на всю простоту, наши примеры помогли нам увидеть потенциальные проблемы, возникающие из-за использования незрелых подходов для асинхронной обработки данных.

Мы познакомились с краткой историей реактивного программирования, чтобы подчеркнуть архитектурные проблемы, для борьбы с которыми предназначалось реактивное программирование. В этом отношении история успеха Netflix наглядно показывает, что такая маленькая библиотека, как RxJava, может стать надежной опорой в бизнесе. Также мы узнали, что на волне успеха RxJava многие компании и проекты создали свои реализации реактивных библиотек, в результате чего появился многообразный реактивный ландшафт. Это многообразие обусловило необходимость разработать реактивный стандарт Reactive Streams, о котором поговорим в следующей главе.

Глава 3

Reactive Streams – новый стандарт потоков

Здесь мы рассмотрим некоторые проблемы, упоминавшиеся в предыдущей главе, а также сложности, возникающие, когда в одном проекте встречается несколько реактивных библиотек. Поближе познакомимся с идеей обратного давления в реактивных системах. Продолжим исследовать решения, предлагаемые библиотекой RxJava, и их ограничения. Посмотрим, как стандарт Reactive Streams решает эти проблемы, изучим его основы. Узнаем, какие изменения в реактивном ландшафте обусловлены этим стандартом. Наконец, чтобы закрепить полученные знания, напишем простое приложение и объединим в нем несколько реактивных библиотек.

В этой главе мы рассмотрим следующие темы:

- проблемы общего API;
- проблемы управления обратным давлением;
- примеры Reactive Streams;
- проблемы совместимости технологий;
- Reactive Streams в JDK 9;
- продвинутые идеи в Reactive Streams;
- укрепление реактивного ландшафта;
- Reactive Streams в действии.

Реактивность для всех

В предыдущих главах мы узнали много интересного о реактивном программировании в Spring, а также о том, какую роль играет в нем RxJava. Поняли, что необходимо использовать приемы реактивного программирования для реализации реактивных систем. Познакомились с кратким обзором реактивного ландшафта и с доступными библиотеке RxJava альтернативами, которые помогают быстро внедрить реактивное программирование в практику.

Проблема несовместимости API

С одной стороны, благодаря множеству библиотек, таких как RxJava, и базовых механизмов Java, таких как `CompletableFuture`, есть возможность выбирать стиль написания кода. Например, для реализации обработки потока элементов можно полагаться на RxJava API, а для организации несложных асинхронных взаимодействий типа «запрос-ответ» – на более подходящий инструмент `CompletableFuture`. Также можно использовать фреймворки с наборами специализированных классов, таких как `org.springframework.util.concurrent.ListenableFuture`, помогающих выполнять асинхронные взаимодействия между компонентами и упрощающих работу с фреймворками.

С другой стороны, обилие вариантов усложняет систему. Например, из-за двух библиотек, реализующих асинхронные неблокирующие взаимодействия, но имеющих разные API, необходимо создавать дополнительный служебный класс для преобразования одного обратного вызова в другой, и наоборот.

```
interface AsyncDatabaseClient {                                // (1)
    <T> CompletableFuture<T> store(CompletableFuture<T> stage); //
}                                                                //

final class AsyncAdapters {
    public static <T> CompletableFuture<T> toCompletion(        // (2)
        ListenableFuture<T> future) {                          //
                                                                //
        CompletableFuture<T> completableFuture =              // (2.1)
            new CompletableFuture<>();                          //
                                                                //
        future.addCallback(                                    // (2.2)
            completableFuture::complete,                       //
            completableFuture::completeExceptionally           //
        );                                                      //
                                                                //
        return completableFuture;                               //
    }                                                            //

    public static <T> ListenableFuture<T> toListenable(         // (3)
        CompletableFuture<T> stage) {                          //
        SettableListenableFuture<T> future =                   // (3.1)
            new SettableListenableFuture<>();                  //
                                                                //
        stage.whenComplete((v, t) -> {                         // (3.2)
            if (t == null) {                                    //
                future.set(v);                                  //
            }                                                    //
        }                                                       //
        else {                                                  //

```

```

        future.setException(t);                //
    }                                           //
});                                           //
//
    return future;                            //
}                                           //
}

@RestController                               // (4)
public class MyController {                  //
    ...                                     //
    @RequestMapping                          //
    public ListenableFuture<?> requestData() { // (4.1)
        AsyncRestTemplate httpClient = ...; //
        AsyncDatabaseClient databaseClient = ...; //
                                                //
        CompletionStage<String> completionStage = toCompletion( // (4.2)
            httpClient.execute(...)           //
        );                                   //
                                                //
        return toListenable(                 // (4.3)
            databaseClient.store(completionStage) //
        );                                   //
    }                                         //
}                                           //

```

Вот как действует этот код.

1. Объявление интерфейса асинхронного клиента базы данных, который является типичным примером возможного интерфейса клиента с асинхронным доступом к базе данных.
2. Метод преобразования экземпляра `ListenableFuture` в `CompletionStage`. В строке (2.1), чтобы обеспечить ручное управление экземпляром `CompletionStage`, создается его прямая реализация `CompletableFuture` вызовом конструктора без аргументов. Для интеграции с `ListenableFuture` нужно добавить обратный вызов (2.2) и в нем напрямую использовать `CompletableFuture` API.
3. Метод преобразования экземпляра `CompletionStage` в `ListenableFuture`. В строке (3.1) объявляется специализированная реализация `ListenableFuture` – `SettableListenableFuture`. Она позволяет вручную передать результат выполнения `CompletionStage` в строке (3.2).
4. Объявление класса REST-контроллера. В строке (4.1) определяется метод обработки запросов, который действует асинхронно и возвращает `ListenableFuture` для обработки результата выполнения неблокирующим способом. В свою очередь, чтобы сохранить результат выполнения

`AsyncRestTemplate`, его нужно преобразовать в `CompletionStage` (4.2). Наконец, чтобы удовлетворить поддерживаемый API, нужно результат сохранения преобразовать в `ListenableFuture` (4.3).

Можем заметить, что не существует прямой интеграции между `ListenableFuture` из Spring Framework 4.x и `CompletionStage`. Кроме того, пример не является исключением из правил в реактивном программировании. Многие библиотеки и фреймворки предлагают свои интерфейсы и классы для организации асинхронных взаимодействий между компонентами, в том числе простые методы типа «запрос-ответ», наряду с инструментами потоковой обработки. Во многих случаях, чтобы решить эту проблему и обеспечить совместимость независимых библиотек, нужно реализовать и использовать свои средства преобразования. Однако такие средства преобразования могут содержать ошибки и требовать дополнительных усилий по сопровождению.



В Spring Framework 5.x интерфейс `ListenableFuture` был дополнен методом `completable` для устранения несовместимости. Узнать больше можно по ссылке <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/concurrent/ListenableFuture.html#completable-->.

Основная проблема заключается в том, что нет единого соглашения, которое позволило бы создателям библиотек приводить свои API к единому знаменателю. Например, как показано в главе 2 «*Реактивное программирование в Spring – основные понятия*», библиотека `RxJava` используется во многих фреймворках, таких как `Vert.x`, `Ratpack`, `Retrofit` и т. д.

Все они предлагают поддержку пользователей `RxJava` и дополнительные модули для упрощения интеграции с существующими проектами. На первый взгляд это замечательно, потому что список проектов, использующих `RxJava 1.x`, обширен, в него входят фреймворки для разработки веб-, настольных и мобильных приложений. Однако поддержка эта таит в себе множество подводных камней.

Первая проблема, которая обычно возникает, когда в одном месте встречаются несколько библиотек, совместимых с `RxJava 1.x`, – это грубая несовместимость версий. Версия `RxJava 1.x` развивалась очень быстро, и многие создатели библиотек не успевали вовремя обновить свои зависимости. Время от времени изменения вызывали множество внутренних изменений, которые в конечном счете стали причиной несовместимости отдельных версий. Как следствие использование разных библиотек и фреймворков, зависящих от разных версий `RxJava 1`, может приводить к нежелательным последствиям.

Вторая проблема похожа на первую. Модификации `RxJava` не стандартизованы. Под *модификациями* в данном случае понимается возможность использования дополнительной реализации `Observable` или этапа преобразования, что является обычным делом при разработке расширений `RxJava`. Из-за отсутствия стандарт-

ного API и быстрого развития внутренних компонентов поддержка модифицированных реализаций стала еще одной большой проблемой.



Отличный пример существенных изменений в версии можно найти по ссылке: <https://github.com/ReactiveX/RxJava/issues/802>.

Модели обмена PULL и PUSH

Наконец, чтобы понять проблему, описанную в предыдущем разделе, вернемся в прошлое и проанализируем исходную модель взаимодействий между источником информации и его подписчиками.

На раннем этапе развития реактивного программирования все библиотеки проектировались с расчетом, чтобы данные принудительно передавались (push) от источника к подписчикам. Такое решение было принято потому, что чистая модель извлечения данных по запросу со стороны подписчика (pull) в некоторых случаях недостаточно эффективна. Примером может служить взаимодействие в сетях с ограниченной пропускной способностью. Допустим, нам нужно отфильтровать гигантский список данных и взять только первые десять элементов. Если для решения задачи использовать модель *PULL* (извлечения по запросу), получим следующий код.

```
final AsyncDatabaseClient dbClient = ... // (1)

public CompletionStage<Queue<Item>> list(int count) { // (2)
    BlockingQueue<Item> storage = new ArrayBlockingQueue<>(count); //
    CompletableFuture<Queue<Item>> result //
        = new CompletableFuture<>(); //
    //
    pull("1", storage, result, count); // (2.1)
    //
    return result; //
} //

void pull( // (3)
    String elementId, //
    Queue<Item> queue, //
    CompletableFuture resultFuture, //
    int count //
) { //
    dbClient.getNextAfterId(elementId).thenAccept(item -> { //
        if (isValid(item)) { // (3.1)
            queue.offer(item); //
            //
            if (queue.size() == count) { // (3.2)
                resultFuture.complete(queue); //
            }
        }
    })
}
```

```
        return; //
    } //
} //
//
pull(item.getId(), // (3.3)
    queue, //
    resultFuture, //
    count); //
}); //
}
```

Вот как действует этот код.

1. Объявление поля `AsyncDatabaseClient`. С помощью данного клиента мы организуем асинхронные, неблокирующие взаимодействия с внешней базой данных.
2. Объявление метода `list`. Здесь мы объявляем асинхронный контракт, возвращающий результат типа `CompletionStage`. В свою очередь, для объединения извлекаемых результатов и асинхронной передачи их вызывающему коду объявляем `Queue` и `CompletableFuture`, чтобы сохранить принятые значения и позднее вручную отправить собранную очередь `Queue`. В строке (2.1) выполняется первый вызов метода `pull`.
3. Объявление метода `pull`. Внутри он вызывает `AsyncDatabaseClient#getNextAfterId`, чтобы асинхронно выполнить запрос и получить результат. После получения результата в строке (3.1) производится фильтрация. Если элемент соответствует критериям, он добавляется в очередь. Кроме того, в строке (3.2) проверяется количество собранных элементов, и, если оно соответствует заданному, элементы посылаются вызывающему коду, и метод завершается. Если ни одно из условий не выполняется, производится рекурсивный вызов метода `pull` (3.3).

Как можно заметить, служба взаимодействует с базой данных асинхронным и неблокирующим способом. Казалось бы, реализация выглядит безупречно. Однако если посмотреть на временную диаграмму (рис. 3.1), можно увидеть проблему.

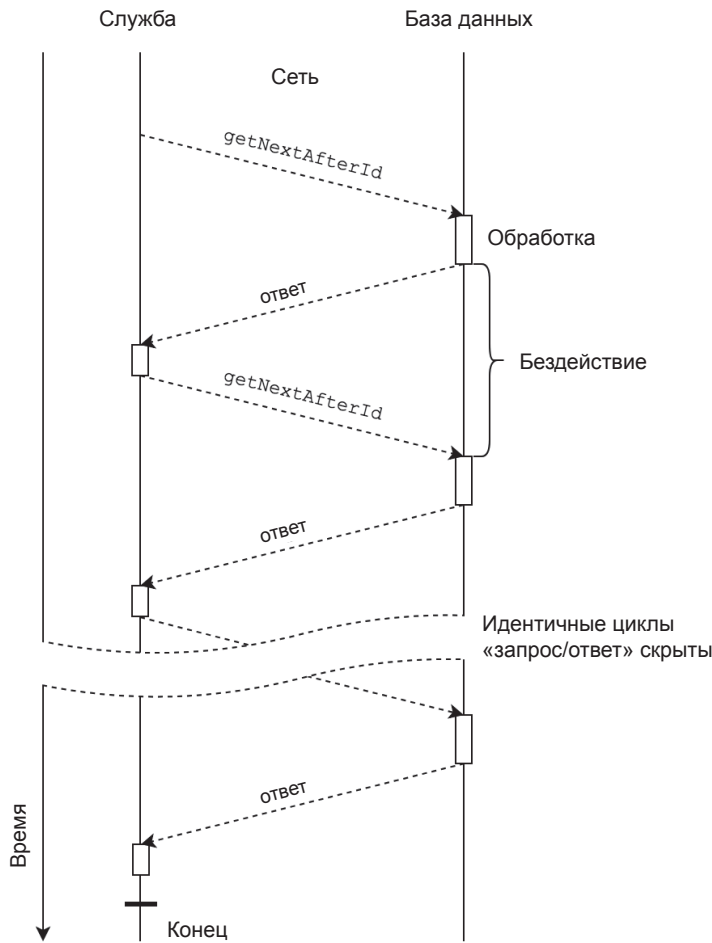


Рис. 3.1. Процесс получения данных по запросу

Как показано на рис. 3.1, запрос и получение элементов по очереди сопряжены с определенными затратами времени. С точки зрения службы большая часть общего времени обработки расходуется на ожидание ответа. Во-первых, доставка запроса в базу данных занимает время, и дополнительная сетевая активность может удвоить или даже утроить срок обработки. Кроме того, базе данных неизвестно число будущих запросов, а значит, она не может сгенерировать данные заранее и потому простаивает в ожидании нового запроса. То есть база данных простаивает, пока ответ доставляется службе, затем служба обрабатывает его и посылает новый запрос.

Для уменьшения общего времени выполнения и увеличения эффективности модель получения данных по запросу можно объединить с пакетной обработкой, как показано в следующем варианте предыдущего примера.


```

void pull(                                     // (1)
    String elementId,                         //
    Queue<Item> queue,                        //
    CompletableFuture resultFuture,         //
    int count                                //
) {                                           //
    dbClient.getNextBatchAfterId(elementId, count) // (2)
        .thenAccept(items -> {              //
            for(Item item : items) {         // (2.1)
                if (isValid(item)) {         //
                    queue.offer(item);       //
                }                             //
            }                                 //
        }                                    //
    }                                         //
    pull(items.get(items.size() - 1)         // (3)
        .getId(),
        queue,
        resultFuture,
        count);
});
}

```

Вот пояснения к этому коду.

1. Объявление того же метода `pull`, как в предыдущем примере.
2. Вызов `getNextBatchAfterId`. Как можно заметить, метод `AsyncDatabaseClient` позволяет запросить определенное число элементов, которые возвращаются в виде `List<Item>`. Обработка данных выполняется почти так же, как прежде, за исключением дополнительного цикла `for`, обрабатывающего отдельные элементы из пакета (2.1).
3. Рекурсивный вызов метода `pull`, извлекающий дополнительные пакеты элементов, если после фильтрации было получено недостаточно элементов.

С одной стороны, запрашивая элементы пакетами, можно существенно повысить производительность метода `list` и уменьшить общее время обработки. С другой стороны, эта модель взаимодействий тоже несовершенна, как можно видеть на рис. 3.2.

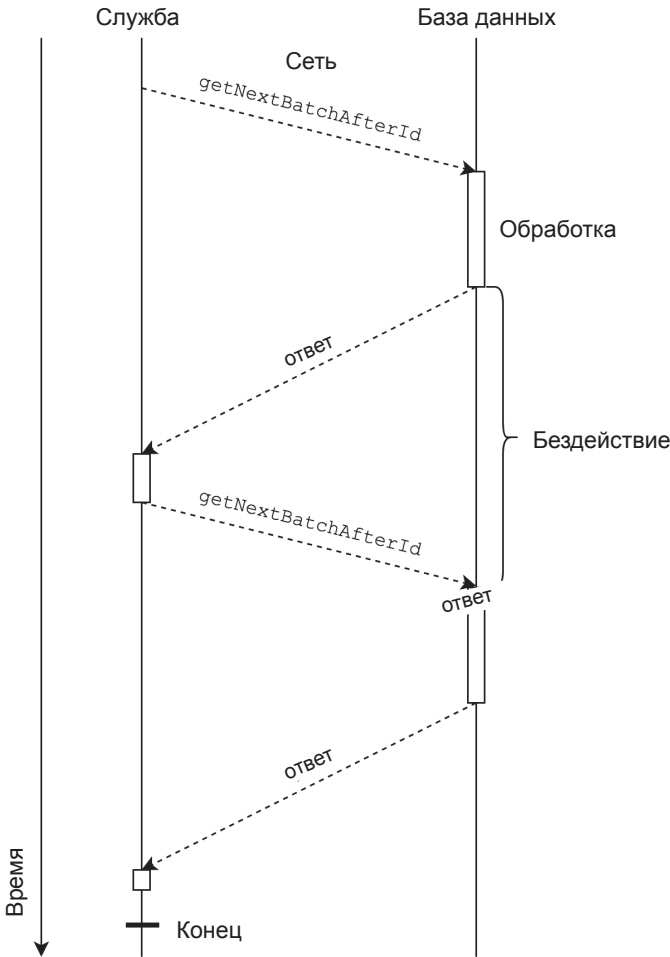


Рис. 3.2. Процесс получения данных пакетами по запросу

Как видите, в данном случае время на обработку тоже расходуется недостаточно эффективно. Например, клиент все еще простаивает, пока база данных обрабатывает запрос. Кроме того, для отправки пакета элементов тоже требуется больше времени. Наконец, последний из дополнительных запросов при пакетной обработке может оказаться фактически избыточным. Например, если останется обработать только один элемент и первый же элемент в следующем пакете удовлетворяет критериям проверки, то остальные элементы в пакете будут отброшены, а значит, время на их извлечение потрачено напрасно.

В качестве заключительной оптимизации можно запросить данные один раз, а источник будет асинхронно передавать (push) их по мере доступности. Следующий фрагмент демонстрирует, как это достигается.

```
public Observable<Item> list(int count) { // (1)
    return dbClient.getStreamOfItems() // (2)
        .filter(item -> isValid(item)) // (2.1)
        .take(count) // (2.2)
} //
```

Пояснения к коду.

1. Объявление метода `list`. Здесь тип `Observable<Item>` возвращаемого значения указывает, что элементы автоматически передаются источником.
2. Запрос потока. Вызывая метод `AsyncDatabaseClient#getStreamOfItems`, мы один раз подписываемся на поток из базы данных. В строке (2.1) осуществляется фильтрация элементов, а оператор `.take()` выбирает объем данных, указанный вызывающим кодом.

Здесь для получения переданных элементов используются классы RxJava 1.x. Сразу после выполнения всех требований посылается сигнал отмены, и соединение с базой данных закрывается. На рис. 3.3 показано, как теперь выглядит процесс обработки.

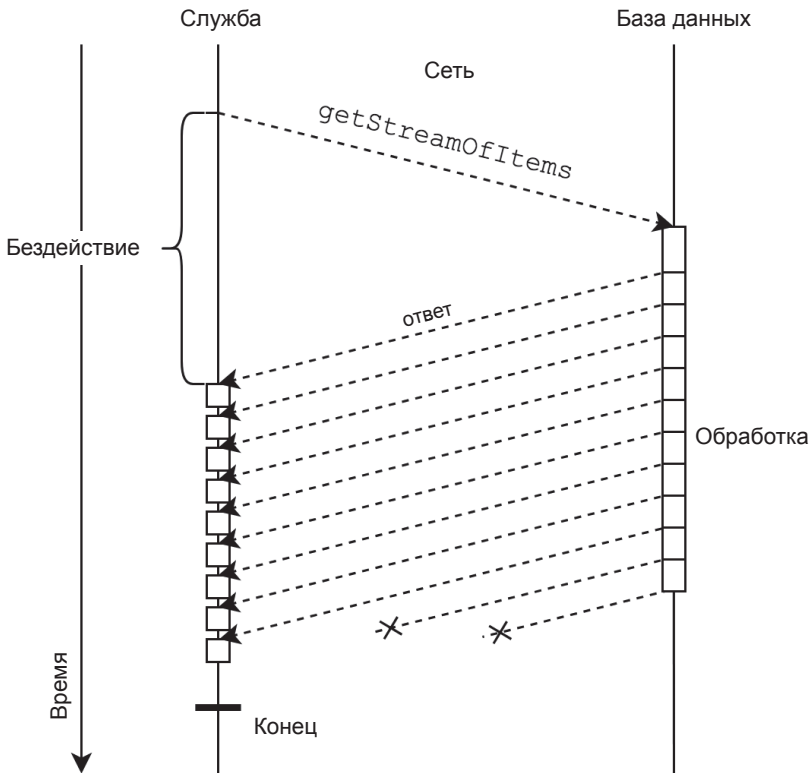


Рис. 3.3. Процесс обработки на основе автоматической передачи данных источником

Как можно судить по диаграмме на рис. 3.3, общее время обработки существенно сократилось. В самом начале у нас есть одно большое окно бездействия, когда служба ожидает получения первого ответа. Затем база данных начинает отправлять последующие элементы по мере их извлечения. В такой ситуации, даже если обработка будет выполняться быстро и заканчиваться раньше, чем прибудет следующий элемент, общее время простоя службы сократится. Однако база данных все еще может генерировать лишние элементы, которые будут проигнорированы службой после получения нужного числа элементов.

Проблема управления потоком данных

С одной стороны, предыдущее объяснение показало, что главной причиной появления модели *PUSH* (асинхронной принудительной передачи данных) была оптимизация общего времени обработки за счет уменьшения количества запросов. Именно поэтому в RxJava 1.x и других похожих библиотеках использовалась модель принудительной передачи данных, и именно поэтому такая потоковая передача данных стала преобладающей в распределенных системах.

С другой стороны, использование одной только модели *PUSH* накладывает некоторые ограничения. Как рассказывалось в главе 1 «Причины выбора Spring», природа взаимодействий на основе обмена сообщениями предполагает, что в ответ на запрос служба получит асинхронный, потенциально бесконечный поток сообщений. Это самый сложный аспект, потому что если производитель не будет учитывать возможностей потребителя, появятся негативные последствия для стабильности всей системы, которые описываются в следующих двух разделах.

Медленный производитель и быстрый потребитель

Начнем с самого простого. Допустим, у нас есть медленный производитель и очень быстрый потребитель. Такая ситуация может возникнуть из-за недостаточности информации о потребителе на стороне производителя.

С одной стороны, такие конфигурации могут быть обусловлены конкретными предположениями. С другой – фактическая среда выполнения может отличаться от случая к случаю, и возможности потребителя могут меняться динамически. Например, у нас всегда есть возможность увеличить число производителей путем масштабирования и тем самым расширить нагрузку на потребителя.

Для решения проблемы важно правильно определить фактическую скорость потребления. К сожалению, чистая модель *PUSH* не обладает такой способностью, следовательно, динамическое управление пропускной способностью системы невозможно.

Быстрый производитель и медленный потребитель

Вторая проблема намного сложнее. Допустим, у нас имеется быстрый производитель и медленный потребитель. Проблема в данном случае заключается в том, что производитель может послать намного больше данных, чем потребитель в состоянии обработать, что может вызвать отказ компонента и катастрофические последствия.

Одним из очевидных решений проблемы является сбор необработанных элементов в очереди, которая может располагаться где-то между производителем и потребителем или даже на стороне потребителя. Такая технология позволит высоконагруженному потребителю не потерять новые данные и обработать их после предыдущего элемента или порции данных.

Одним из критических факторов обработки передаваемых данных с помощью очереди является выбор очереди с надлежащими характеристиками. Очереди делятся на три общих типа. Рассмотрим их в следующих подразделах.

Неограниченная очередь

Первое и самое очевидное решение – использовать очередь неограниченного размера или просто неограниченную очередь. В этом случае все производимые элементы сначала сохраняются внутри очереди, а затем извлекаются фактическим подписчиком. Диаграмма Marble на рис. 3.4 показывает, как протекают взаимодействия в этом случае.

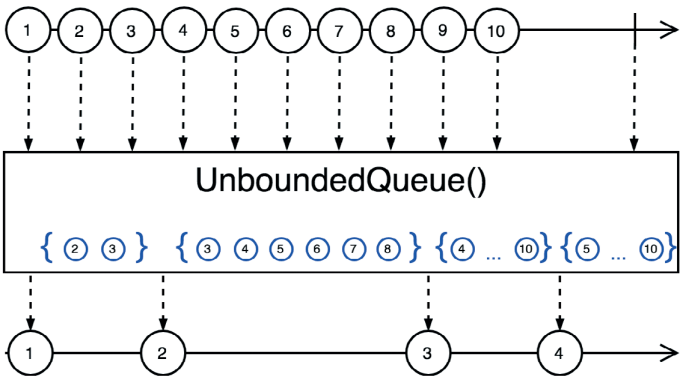


Рис. 3.4. Решение с неограниченной очередью

Главным преимуществом неограниченной очереди является гарантия доставки всех сообщений. То есть потребитель рано или поздно обработает все элементы, хранящиеся в очереди.

С другой стороны, из-за гарантий доставки сообщений снижается устойчивость приложения, потому что на самом деле компьютеры не обладают неограничен-

ными ресурсами. Например, система может аварийно завершиться, исчерпав доступную память.

Ограниченная очередь со сбросом избыточных элементов

Чтобы избежать исчерпания памяти, можно использовать очередь, которая после заполнения будет игнорировать входящие сообщения. Диаграмма Marble на рис. 3.5 изображает очередь, вмещающую два элемента и отбрасывающую следующие элементы.

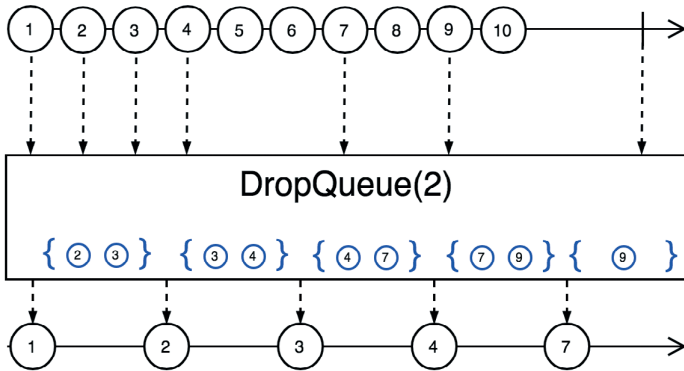


Рис. 3.5. Пример ограниченной очереди, вмещающей два элемента

В целом это решение учитывает ограниченность ресурсов и позволяет настраивать емкость очереди, исходя из имеющихся возможностей. Очереди подобного типа обычно используются, когда важность сообщений низка. Примером может служить поток событий изменения набора данных. Каждое такое событие запускает некоторый механизм пересчета статистик, использующий весь набор данных и выполняющийся достаточно долго, если сравнивать с частотой поступления событий. Единственное, что здесь важно, – сам факт изменения набора данных, а какие данные при этом изменились, несущественно.



В предыдущем примере подразумевается простейшая стратегия удаления самого нового элемента. Однако, вообще говоря, есть несколько стратегий выбора элемента для удаления. Например, удаление по приоритету, удаление самого старого элемента и т. д.

Ограниченная очередь с блокировкой

Если важность сообщений слишком высока, решение на основе ограниченной очереди со сбросом избыточных элементов может оказаться неприемлемым. Пример – платежная система, которая должна обработать каждый платеж. В такой ситуации вместо удаления избыточных элементов можно просто блокировать производителя по достижении предела. Очереди, обладающие возможностью блокировки производителя, называются очередями с блокировкой, или бло-

кирующими очередями. Пример использования блокирующей очереди емкостью на три элемента представлен на рис. 3.6.

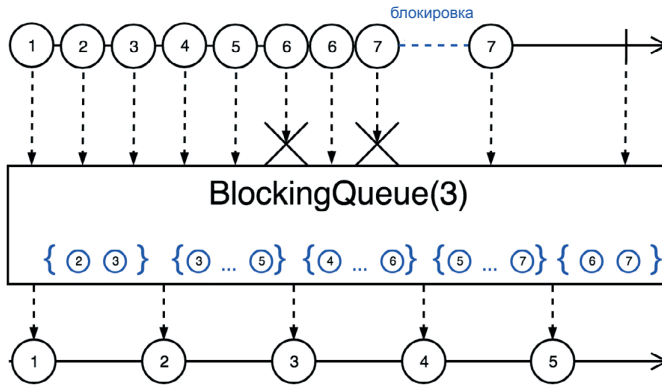


Рис. 3.6. Пример ограниченной очереди с блокировкой, вмещающей три элемента

К сожалению, такой подход лишает систему преимуществ асинхронной обработки. Когда производитель достигнет предела, он будет заблокирован и останется в таком состоянии, пока потребитель не извлечет элемент и не освободит место в очереди. Нетрудно догадаться, что пропускная способность всей системы будет определяться пропускной способностью самого медленного потребителя. Однако этот подход не только лишает нас преимуществ асинхронного выполнения, но и снижает эффективность использования ресурсов. Следовательно, ни одно из перечисленных решений не подходит, если мы хотим добиться наличия всех трех характеристик: устойчивости, эластичности и отзывчивости.

Кроме того, создание очередей может усложнить общий дизайн системы и потребовать найти компромисс между упомянутыми решениями, что само по себе еще одна проблема.

Итак, неуправляемая семантика чистой модели *PUSH* вызывает множество нежелательных ситуаций. Вот почему в манифесте реактивных систем особо отмечается важность механизма, позволяющего системам реагировать на нагрузку, или, другими словами, механизма управления обратным давлением.

К сожалению, реактивные библиотеки, такие как RxJava 1.x, не предлагают стандартного механизма. То есть не существует явного API, который позволил бы организовать управление обратным давлением «из коробки».



Следует отметить, что в чистой модели *PUSH* скорость отправки данных можно стабилизировать, используя приемы пакетной обработки. Библиотека RxJava 1.x предлагает для этого такие операторы, как `.window` и `.buffer`, которые позволяют собрать элементы за определенный интервал времени в подпоток или коллекцию соответственно. Примером, где такой прием дает прирост производительности, может служить па-

кетная вставка или пакетное изменение информации в базе данных. К сожалению, не все службы поддерживают пакетные операции. То есть этот прием имеет ограниченное применение.

Решение

В конце 2013 года собралась группа гениальных инженеров из компаний Lightbend, Netflix и Pivotal, чтобы обсудить описанную проблему и представить решение сообществу JVM. После года напряженной работы мир увидел первый проект стандарта Reactive Streams. В нем не содержалось ничего необычного – основная идея заключалась в стандартизации уже знакомых шаблонов реактивного программирования, с которыми мы познакомились в предыдущей главе. Рассмотрим ее подробнее в следующем разделе.

Основные положения стандарта Reactive Streams

Стандарт Reactive Streams определяет четыре основных интерфейса: Publisher, Subscriber, Subscription и Processor. Поскольку инициатива развивается вне рамок какой-либо организации, она стала доступна в виде отдельного файла JAR, где интерфейсы находятся в пакете `org.reactivestreams`.

В целом указанные интерфейсы похожи на те, что мы видели выше (например, в RxJava 1.x). В некотором смысле они отражают хорошо известные классы из RxJava. Первые два интерфейса напоминают классы Observable/Observer, реализующие классическую модель «Издатель/Подписчик». Соответственно, они получили имена Publisher и Subscriber. Чтобы убедиться в сходстве этих интерфейсов и классов Observable и Observer, рассмотрим их объявления.

```
package org.reactivestreams;

public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
```

Это объявление интерфейса Publisher. Как видите, он определяет только один метод, предназначенный для регистрации подписчика Subscriber. В отличие от класса Observable, который проектировался как реализация предметно-ориентированного языка (DSL), интерфейс Publisher предлагает лишь стандартную точку для создания соединения между издателем Publisher и подписчиком Subscriber. В отличие от интерфейса Publisher, интерфейс Subscriber имеет более детализированный API, почти идентичный интерфейсу класса Observer из RxJava.


```
package org.reactivestreams;

public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
```

Как можно заметить, кроме трех методов, идентичных методам класса `Observer` из `RxJava`, стандарт определяет новый дополнительный метод `onSubscribe`.

Этот концептуально новый метод дает стандартную возможность сообщить подписчику `Subscriber` об успешной подписке. Кроме того, его входной параметр представляет новый контракт `Subscription`. Чтобы понять его суть, рассмотрим соответствующий интерфейс.

```
package org.reactivestreams;

public interface Subscription {
    void request(long n);
    void cancel();
}
```

Как видите, подписка `Subscription` определяет основу для управления производством элементов. По аналогии с `Subscription#unsubscribe()` в `RxJava 1.x` здесь имеется метод `cancel()`, позволяющий отменить подписку на поток или даже публикацию событий в целом. Однако самое значительное усовершенствование, пришедшее вместе с функцией отмены, – это новый метод `request`. Стандарт `Reactive Stream` предлагает использовать его для расширения возможностей взаимодействий между издателем `Publisher` и подписчиком `Subscriber`. Теперь подписчик `Subscriber` может сообщить издателю `Publisher`, сколько данных тот должен послать, передав аргумент в вызов метода `request`, и быть в полной уверенности, что число входящих элементов не превысит ограничения. Рассмотрим диаграмму `Marble` на рис. 3.7, чтобы понять, как действует этот механизм.

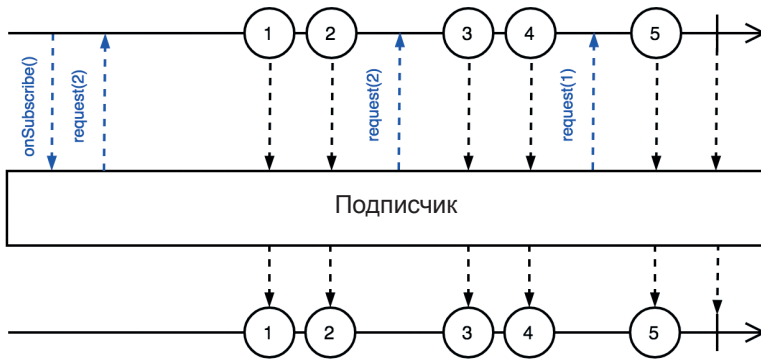


Рис. 3.7. Механизм обратного давления

Как показано на рис. 3.7, теперь издатель *Publisher* гарантирует, что новая порция данных будет отправлена только в ответ на запрос подписчика *Subscriber*. Конкретное поведение издателя зависит от фактической реализации *Publisher*, которая может варьироваться от простого ожидания с блокировкой до сложного механизма генерации данных только в ответ на запрос подписчика. В любом случае, имея такие гарантии, мы можем не задумываться об организации дополнительных очередей.

Кроме того, помимо простой модели *PUSH* стандарт предлагает гибридную модель *PUSH-PULL*, которая поддерживает возможность управления обратным давлением.

Чтобы понять широту возможностей гибридной модели, вернемся к предыдущему примеру потоковой передачи информации из базы данных и сравним этот прием с предыдущим.

```
public Publisher<Item> list(int count) { // (1)
    Publisher<Item> source = dbClient.getStreamOfItems(); // (2)
    TakeFilterOperator<Item> takeFilter =
        new TakeFilterOperator<>( // (2.1)
            source, //
            count, //
            item -> isValid(item) //
        ); //
    return takeFilter; // (3)
}
```

Вот пояснения к этому коду.

1. Объявление метода `list`. Здесь мы следуем стандарту Reactive Streams и возвращаем интерфейс `Publisher<>`.

2. Вызов метода `AsyncDatabaseClient#getStreamOfItems`. Используем обновленный метод, который возвращает `Publisher<>`. В строке (2.1) определяется собственная реализация операторов `Take` и `Filter`, которая принимает число элементов для загрузки. Дополнительно передается наша реализация `Predicate`, осуществляющая проверку элементов в потоке.
3. В этой точке возвращаем только что созданный экземпляр `TakeFilterOperator`. Даже при том что этот оператор имеет другой тип, он все же расширяет интерфейс `Publisher`.

Ниже приводится код с реализацией нашего оператора `TakeFilterOperator`, знать и понимать которую совершенно необходимо для данного примера.

```
public class TakeFilterOperator<T> implements Publisher<T> {    // (1)
    ...                                                         //

    public void subscribe(Subscriber s) {                        // (2)
        source.subscribe(new TakeFilterInner<>(s, take, predicate)); //
    }                                                            //

    static final class TakeFilterInner<T>
        implements Subscriber<T>,                               // (3)
        Subscription {
            final Subscriber<T> actual;                          //
            final int take;                                       //
            final Predicate<T> predicate;                        //
            final Queue<T> queue;                                 //
            Subscription current;                                 //
            int remaining;                                        //
            int filtered;                                         //
            volatile long requested;                             //
            ...                                                  //

            TakeFilterInner(
                Subscriber<T> actual,                            // (4)
                int take,
                Predicate<T> predicate
            ) { ... }                                           //

            public void onSubscribe(Subscription current) {    // (5)
                ...                                             //
                current.request(take);                          // (5.1)
                ...                                             //
            }                                                    //

            public void onNext(T element) {                    // (6)
                ...                                             //
                long r = requested;                             //
            }
        }
    }
}
```

```

Subscriber<T> a = actual; //
Subscription s = current; //

if (remaining > 0) { // (7)
    boolean isValid = predicate.test(element); //
    boolean isEmpty = queue.isEmpty(); //
    if (isValid && r > 0 && isEmpty) { //
        a.onNext(element); // (7.1)
        remaining--; //
        ... //
    } //
    else if (isValid && (r == 0 || !isEmpty)) { //
        queue.offer(element); // (7.2)
        remaining--; //
        ... //
    } //
    else if (!isValid) { //
        filtered++; // (7.3)
    } //
} //
else { // (7.4)
    s.cancel(); //
    onComplete(); //
} //

if (filtered > 0 && remaining / filtered < 2) { // (8)
    s.request(take); //
    filtered = 0; //
} //

}
... // (9)
}
}

```

Вот пояснения к коду.

1. Объявление класса `TakeFilterOperator`. Этот класс расширяет интерфейс `Publisher<>`. Кроме того, за многоточием скрытаны конструктор и поля класса.
2. Реализация метода `Subscriber#subscribe`. Как можно заключить из этой реализации, для поддержки дополнительной логики потока мы должны завернуть фактическую реализацию `Subscriber` в класс-адаптер, реализующий тот же интерфейс.
3. Объявление класса `TakeFilterOperator.TakeFilterInner`. Он реализует интерфейс `Subscriber` и играет самую важную роль в примере, потому что передается издателю в качестве фактического подписчика `Subscriber`.

После передачи элемента в `onNext` он фильтруется и передается нижестоящему подписчику `Subscriber`. Помимо интерфейса `Subscriber` класс `TakeFilterInner` реализует также интерфейс `Subscription`, что позволяет передать его нижестоящему подписчику `Subscriber` и соответственно контролировать все его требования. Обратите внимание, что здесь очередь `Queue` является экземпляром `ArrayBlockingQueue`, размер которой равен значению поля `take`. Прием с использованием вложенного класса, расширяющего сразу два интерфейса, `Subscriber` и `Subscription`, является классическим способом реализации промежуточного этапа преобразования.

4. Объявление конструктора. Как можно заметить, помимо параметров `take` и `predicate`, мы принимаем экземпляр `actual` фактического подписчика, который был подписан на `TakeFilterOperator` вызовом метода `subscribe()`.
5. Реализация метода `Subscriber#onSubscribe`. Наибольший интерес представляет строка (5.1). Здесь производится первый вызов метода `Subscription#request`, который посылает запрос удаленной базе данных, что обычно происходит в первом вызове метода `onSubscribe`.
6. Определение метода `Subscriber#onNext`, принимающего очередной элемент для обработки.
7. Процесс обработки элемента. В нем имеется четыре ключевые точки. Если число элементов `remaining`, которые необходимо извлечь, больше нуля, фактический подписчик `Subscriber` запросил данные (допустим, элемент) и в очереди отсутствуют другие элементы, то этот элемент можно послать непосредственно нижестоящему подписчику (7.1). Если данные не были запрошены или в очереди имеются другие элементы, мы должны поместить новый элемент в очередь (чтобы сохранить порядок их следования) и доставить его позже (7.2). Если элемент недопустим, мы увеличиваем счетчик `filtered` отфильтрованных элементов (7.3). Наконец, если число `remaining` равно нулю, мы должны отменить подписку вызовом метода `cancel` (7.4) и закрыть поток.
8. Механизм запроса дополнительных данных. Если число `filtered` отфильтрованных элементов достигло установленного предела, запрашиваем дополнительную порцию данных, не блокируя весь процесс.
9. Реализация остальных методов `Subscriber` и `Subscriptions`.

Как правило, когда соединение с базой данных будет установлено и экземпляр `TakeFilterOperator` получит подписку `Subscription`, базе данных отправляется первый запрос с заданным числом элементов. Сразу после этого база данных начинает генерировать указанное число элементов и передавать их по мере готовности. Логика `TakeFilterOperator` определяет случай, когда должна запрашиваться следующая порция данных. Когда это происходит, служба посылает базе данных новый неблокирующий запрос на следующую порцию данных.

Обратите внимание: стандарт Reactive Streams явно указывает, что в методе `Subscription#request` не рекомендуется использовать операции, которые могут заблокировать вызывающий поток.



За дополнительной информацией обращайтесь по ссылке <https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md#3.4>.

Диаграмма на рис. 3.8 демонстрирует, как протекают взаимодействия между службой и базой данных.

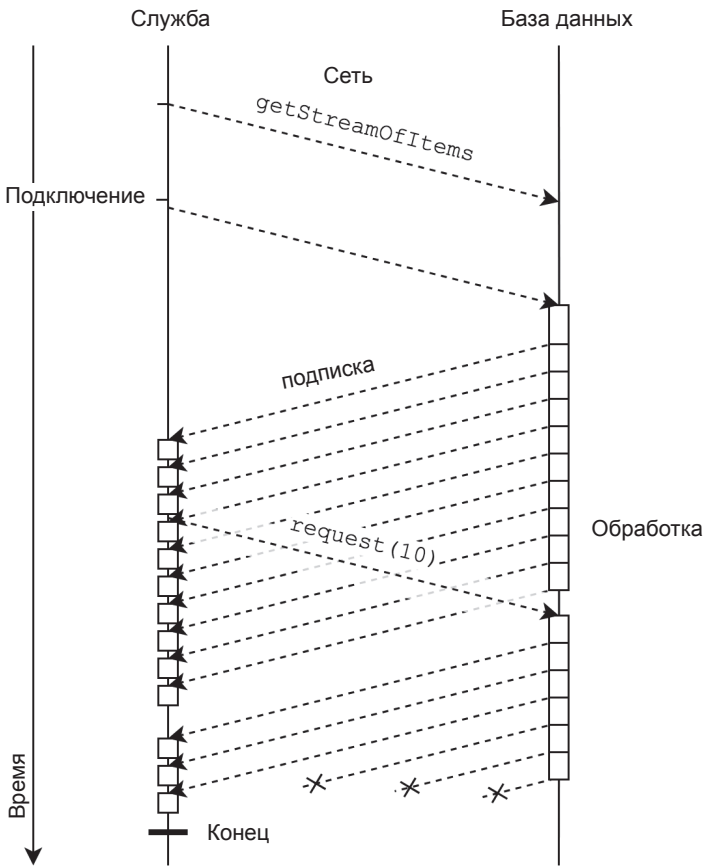


Рис. 3.8. Порядок работы гибридной модели PUSH-PULL

Как показано на рис. 3.8, первый элемент может поступить из базы данных чуть позже из-за особенностей контракта взаимодействия между `Publisher` и `Subscriber`, определяемого стандартом Reactive Streams. Запрос новой порции данных не требует прерывать или блокировать процесс обработки элементов. Со-

ответственно, общее время обработки почти не изменяется.

Однако в некоторых случаях предпочтительнее использовать чистую модель *PUSH*. К счастью, стандарт Reactive Streams обладает достаточной гибкостью. Наряду с динамической моделью *PUSH-PULL* он предлагает отдельные модели *PUSH* и *PULL*. Согласно документации, чтобы получить чистую модель *PUSH*, можно запросить $2^{63} - 1$ элементов (`java.lang.Long.MAX_VALUE`).



Это число можно считать безразмерным, потому что на текущем уровне развития техники невозможно выполнить требование на отправку $2^{63} - 1$ элементов за разумное время (если отправлять один элемент каждую наносекунду, на выполнение запроса потребуется 292 года). Соответственно, реализации `Publisher` разрешается прекратить слежение за количеством отправленных элементов при получении такого требования (<https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md#3.17>).

А чтобы переключиться на чистую модель *PULL*, достаточно запрашивать только один элемент в каждом вызове `Subscriber#onNext`.

Требования Reactive Streams в действии

В целом, как следует из предыдущего раздела, стандарт Reactive Streams определяет относительно простые интерфейсы, но общая идея достаточно сложна. Поэтому далее детально изучим на типичном примере центральную идею и поведение трех интерфейсов.

Возьмем в качестве такого примера подписку на рассылку новостей и посмотрим, как сделать ее более интеллектуальной, используя новые интерфейсы Reactive Streams. Взгляните на следующий код, создающий объект издателя `Publisher` для службы рассылки новостей.

```
NewsServicePublisher newsService = new NewsServicePublisher();
```

Теперь создадим экземпляр подписчика `Subscriber` и подпишем его на новости в `NewsService`.

```
NewsServiceSubscriber subscriber = new NewsServiceSubscriber(5);  
newsService.subscribe(subscriber);  
...  
subscriber.eventuallyReadDigest();
```

Вызовом метода `subscribe()` экземпляра `newsService` мы изъявляем желание получать последние новости. Обычно перед отправкой краткого изложения новостей хорошая служба посылает приветственное письмо с информацией о подписке и порядке действий для ее отмены. Это действие точно вписывается в наш ме-

тод `Subscriber#onSubscribe()`, который информирует подписчика `Subscriber` об успешной подписке и дает ему возможность отменить подписку. Поскольку наша служба следует правилам стандарта Reactive Streams, клиент может выбрать столько новостей, сколько сможет прочитать. Только после того как клиент укажет количество новостей в первой порции вызовом `Subscription#request`, служба начнет посылать новости вызовом метода `Subscriber#onNext`.

В реальной жизни мы можем отложить чтение новостей до вечера или до конца недели и вручную проверить ящик входящих сообщений. На стороне подписчика эта логика реализуется методом `NewsServiceSubscriber#eventuallyReadDigests()`. В общем и целом такое поведение означает, что в почтовом ящике пользователя накапливаются новости, и служба с обычной моделью подписки легко может переполнить почтовые ящики подписчиков.

Когда служба новостей бездумно посылает сообщения подписчику, который, в свою очередь, не торопится прочитать их, поставщик услуг электронной почты обычно помещает службу новостей в черный список. В таком случае подписчик рискует пропустить важную новость. Даже если этого не произойдет, подписчик едва ли будет доволен, столкнувшись с необходимостью разбирать содержимое почтового ящика, заполненного сообщениями от службы новостей. Поэтому, чтобы не вызвать недовольство подписчика, служба новостей должна определить некоторую стратегию отправки новостей. Допустим, факт чтения новостей может использоваться как подтверждение продолжения обслуживания. В таком случае мы можем предусмотреть логику, которая будет посылать новые сообщения, только убедившись, что все предыдущие новости прочитаны. Этот механизм легко реализовать, руководствуясь стандартом. Следующий фрагмент кода демонстрирует пример реализации такого механизма.

```
class NewsServiceSubscriber implements Subscriber<NewsLetter> { // (1)
    final Queue<NewsLetter> mailbox = new ConcurrentLinkedQueue<>(); //
    final int take; //
    final AtomicInteger remaining = new AtomicInteger(); //
    Subscription subscription; //

    public NewsServiceSubscriber(int take) { ... } // (2)

    public void onSubscribe(Subscription s) { // (3)
        ... //
        subscription = s; //
        subscription.request(take); // (3.1)
        ... //
    } //

    public void onNext(NewsLetter newsLetter) { // (4)
        mailbox.offer(newsLetter); //
    } //
```



```

public void onError(Throwable t) { ... }           // (5)
public void onComplete() { ... }                  //

public Optional<NewsLetter> eventuallyReadDigest() { // (6)
    NewsLetter letter = mailbox.poll();           // (6.1)
    if (letter != null) {                         //
        if (remaining.decrementAndGet() == 0) {  // (6.2)
            subscription.request(take);          //
            remaining.set(take);                  //
        }                                         //
        return Optional.of(letter);              // (6.3)
    }                                             //
    return Optional.empty();                     // (6.4)
}
}
}

```

Пояснения к коду.

1. Объявление класса `NewsServiceSubscriber`, который реализует интерфейс `Subscriber<NewsLetter>`. В нем определяется несколько полезных полей (таких как `mailbox` – почтовый ящик, представленный экземпляром очереди `Queue`, и `subscription`), представляющих подписку, иначе говоря, соглашение между клиентом и службой новостей.
2. Конструктор `NewsServiceSubscriber`. Он принимает один параметр `take`, определяющий количество новостей, которое пользователь готов принять одновременно или в ближайший период.
3. Реализация метода `Subscriber#onSubscribe`. В строке (3.1) вслед за сохранением полученного экземпляра подписки `Subscription` мы посылаем серверу запрос, указывая количество сообщений, которое готов принять пользователь.
4. Реализация метода `Subscriber#onNext`. Логика обработки вновь полученного сообщения довольно проста – она всего лишь помещает его в почтовый ящик `Queue`.
5. Объявление методов `Subscriber#onError` и `Subscriber#onComplete`. Они вызываются сразу после отмены подписки.
6. Объявление общедоступного метода `eventuallyReadDigest`. Прежде всего, чтобы показать, что почтовый ящик может быть пустым, мы указываем тип `Optional` возвращаемого значения. Первым делом в строке (6.1) метод пытается получить из почтового ящика последние непрочитанные новости. Если непрочитанных новостей нет, возвращается значение `Optional.empty()` (6.4). Если в ящике имеются доступные новости, мы уменьшаем счетчик (6.2), представляющий число непрочитанных новостей, запрошенных и полученных от службы. Если получены не все запрошенные новости,

возвращаем заполненный экземпляр `Optional`. Иначе дополнительно посылаем запрос на получение новой порции данных и переустанавливаем счетчик новых сообщений (6.3).

Согласно стандарту, первым вызывается метод `onSubscribe()`, который сохраняет экземпляр `Subscription` локально и сообщает издателю `Publisher` о готовности получать новости вызовом метода `request()`. Далее, получив первое сообщение, он сохраняет его в очереди для чтения в будущем, как это обычно происходит в реальном мире. После того как подписчик прочитает все новости из очереди, издатель `Publisher` уведомляется об этом и может прислать следующую порцию новостей. Далее, если служба новостей изменит политику подписки (в некоторых случаях это отмена текущей подписки), подписчик уведомляется об этом вызовом метода `onComplete`. Затем клиенту предложат ознакомиться с новыми условиями, и если они будут приняты, подписка возобновится автоматически.

Примером ситуации, когда вызывается метод `onError`, может служить случайное (конечно же) удаление из базы данных информации о пользовательских предпочтениях. Сие может быть расценено как сбой, и подписчик получит письмо с извинениями и предложением оформить подписку заново. Наконец, реализация `eventuallyReadDigest` – это просто имитация действий пользователя, таких как открытие почтового ящика, проверка новых сообщений, извлечение писем и маркировка их как прочитанных или просто закрытие почтового ящика, если нет новых сообщений.

Как видите, Reactive Streams подходит для решения проблем. Реализовав такой механизм, мы можем избавить подписчиков от лишних хлопот и не дать службе новостей попасть в черный список поставщика услуг электронной почты.

Введение в понятие обработчика `Processor`

Мы познакомились с тремя основными интерфейсами, определяемыми стандартом Reactive Streams. Мы также видели, как предлагаемый стандартом механизм может повысить качество службы новостей, посылающей сообщения по электронной почте. Однако в начале раздела говорилось, что стандарт определяет четыре основных интерфейса. Последний интерфейс является комбинацией `Publisher` и `Subscriber` и называется `Processor`. Взгляните на следующий код с объявлением этого интерфейса.

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

В отличие от `Publisher` и `Subscriber`, которые по определению являются начальной и конечной точками, интерфейс `Processor` предназначен для добавления некоторых этапов обработки между `Publisher` и `Subscriber`. Поскольку `Processor`

может определять некоторую логику преобразования, он упрощает понимание поведения потокового конвейера и бизнес-логики. Ярким примером использования Processor служит любая бизнес-логика, которая может быть описана нестандартным оператором или обеспечивает дополнительное кеширование потоковых данных. Чтобы получить более полное представление о применении интерфейса Processor, посмотрим, как с его помощью можно усовершенствовать NewsServicePublisher.

Самая простая логика, которая скрывается за NewsServicePublisher, – это доступ к базе данных для подготовки писем и последующей их рассылки.

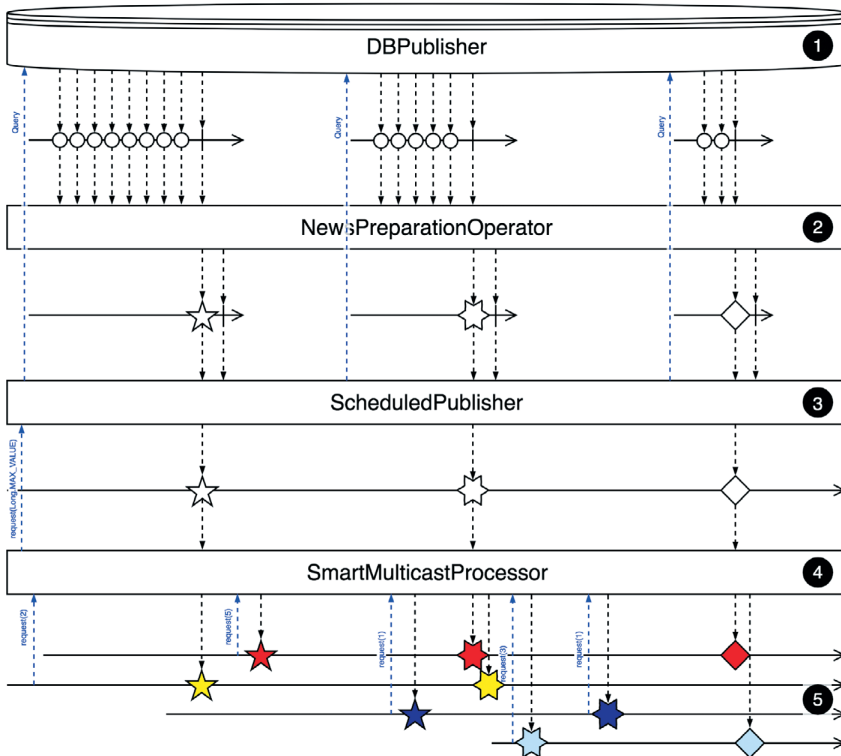


Рис. 3.9. Пример организации процесса рассылки в службе новостей

В этом примере издатель NewsServicePublisher разбит на четыре компонента.

1. Компонент DBPublisher. Отвечает за извлечение новостей из базы данных.
2. Компонент NewsPreparationOperator. Выполняет промежуточное преобразование, объединяя все сообщения, и затем, когда от основного источника поступит сигнал завершения, компонует новости в письмо. Обратите внимание, что этот оператор всегда возвращает один элемент, что соответствует природе операции объединения. Объединение предполагает использова-

ние некоторого хранилища, которым может быть очередь или любая другая коллекция.

3. Компонент `ScheduledPublisher`. Отвечает за планирование периодических задач. В ранее упомянутом случае запланированная задача – это запрос к базе данных (`DBPublisher`), обработка результата и объединение полученных данных в нисходящий поток. Обратите внимание, что `ScheduledPublisher` фактически является бесконечным потоком – он игнорирует завершение объединенного издателя `Publisher`. При отсутствии запросов от нисходящего потока издатель `Publisher` генерирует исключение и передает его фактическому подписчику `Subscriber` вызовом метода `Subscriber#onError`.
4. Компонент `SmartMulticastProcessor`. Этот обработчик `Processor` играет важную роль в потоке. Прежде всего он кеширует последний обзор. Кроме того, данный компонент поддерживает многоадресную рассылку, а значит, нет необходимости создавать один и тот же поток для каждого подписчика `Subscriber` в отдельности. Также, как упоминалось выше, `SmartMulticastProcessor` включает механизм интеллектуального слежения за рассылкой и рассылает новости только тем, кто прочитал предыдущий обзор.
5. Фактические подписчики, то есть экземпляры `NewsServiceSubscriber`.

Диаграмма на рис. 3.9 показывает, что может скрываться за фасадом простого `NewsServicePublisher`. На ней изображен действующий вариант применения интерфейса `Processor`. Как можно заметить, здесь есть три этапа преобразования, но только один должен реализовать интерфейс `Processor`.

Прежде всего, когда необходимо простое преобразование из *A* в *B*, нет нужды использовать интерфейс, одновременно представляющий издателя `Publisher` и подписчика `Subscriber`. Присутствие интерфейса `Subscriber` означает, что, после того как реализация `Processor` подпишется на вышестоящий поток, элементы будут поступать в метод `Subscriber#onNext` и теоретически могут теряться из-за отсутствия нижестоящего подписчика `Subscriber`. Поэтому при использовании такого подхода необходимо оформить подписку в `Processor` до того, как тот подпишется на получение событий от основного издателя `Publisher`.

Это усложняет бизнес-логику и препятствует созданию многократно используемого оператора. Кроме того, при создании реализации `Processor` требуется приложить дополнительные усилия для организации независимого (от основного издателя `Publisher`) управления подписчиком `Subscriber` и правильной реализации механизма обратного давления (например, с использованием очереди, если потребуется). Как следствие снижается производительность или просто уменьшается пропускная способность всего потока из-за неоправданно сложной реализации обработчика `Processor` в роли простого оператора.

Поскольку нужно всего лишь преобразовать *A* в *B*, мы должны просто запустить процесс, когда фактический подписчик `Subscriber` вызовет `Publisher#subscribe`,

и нам совсем не нужна избыточно сложная внутренняя реализация. Таким требованиям удачно соответствует комбинация из нескольких экземпляров `Publisher`, которая просто принимает вышестоящий поток как параметр конструктора и реализует логику адаптера.

С другой стороны, интерфейс `Processor` предстает во всем своем блеске, когда требуется организовать многоадресную рассылку независимо от наличия подписчиков. Он также поддерживает некоторые виды трансформации, так как реализует интерфейс `Subscriber`, который поддерживает такие трансформации, как кеширование.

Учитывая, что мы уже видели реализацию оператора `TakeFilterOperator` и класса `NewsServiceSubscriber`, можно смело утверждать, что большинство экземпляров `Publisher`, `Subscriber` и `Processor` имеют внутреннее устройство, похожее на предыдущие примеры. Поэтому не будем вдаваться в детали реализации каждого класса и рассмотрим только окончательный состав всех компонентов.

```
Publisher<Subscriber<NewsLetter>> newsLetterSubscribersStream = ... // (1)
ScheduledPublisher<NewsLetter> scheduler =                          //
    new ScheduledPublisher<>()                                         //
        () -> new NewsPreparationOperator                           //
            (new DBPublisher(...), ...),                             // (1.1)
            1, TimeUnit.DAYS                                         //
    );                                                                //
SmartMulticastProcessor processor =                                  //
    new SmartMulticastProcessor();                                    //

scheduler.subscribe(processor);                                     // (2)

newsLetterSubscribersStream.subscribe(new Subscriber<>() {          // (3)
    ...                                                              //
    public void onNext(Subscriber<NewsLetter>> s) {                  //
        processor.subscribe(s);                                     // (3.1)
    }                                                                //
    ...                                                              //
});                                                                    //
```

Пояснения к коду.

1. Объявления издателей, оператора и обработчика. Здесь `newsLetterSubscribersStream` представляет бесконечный поток пользователей, подписавшихся на рассылку. В строке (1.1) объявляется `Supplier<? extends Publisher<NewsLetter>>`, который предоставляет экземпляр `DBPublisher`, завернутый в `NewsPreparationOperator`.
2. Здесь `SmartMulticastProcessor` подписывается на `ScheduledPublisher<NewsLetter>`. Эта операция немедленно запускает планировщика, кото-

рый, в свою очередь, подписывается на рассылку внутреннего издателя Publisher.

3. Подписка `newsLetterSubscribersStream`. Объявляется анонимный класс для реализации `Subscriber`. В строке (3.1) мы подписываем каждый новый экземпляр `Subscriber` на `processor`, который осуществляет рассылку новостей всем подписчикам.

В этом примере мы свели все обработчики в одну цепочку, последовательно заворачивая их друг в друга или заставляя компоненты подписываться друг на друга.



В общем случае реализация связки `Publisher/Processor` – сложная задача, поэтому мы не будем подробно описывать реализацию упомянутых операторов или источников в этой главе. Узнать больше о подводных камнях, шаблонах и этапах, которые необходимо реализовать в своей версии `Publisher`, вы сможете по ссылке <https://medium.com/@olehdokuka/mastering-own-reactive-streams-implementation-part-1-publisher-e8eaf928a78c>.

Итак, мы рассмотрели основы стандарта `Reactive Streams`. Познакомились со способом преобразования идей реактивного программирования, выраженных в таких библиотеках, как `RxJava`, в стандартный набор интерфейсов. Попутно узнали, что эти интерфейсы позволяют легко определить модель асинхронных и неблокирующих взаимодействий между компонентами системы. Наконец, опираясь на стандарт `Reactive Streams`, мы можем создать реактивную систему на уровне не только архитектуры всего приложения, но и компонентов меньшего размера.

Проверка совместимости с `Reactive Streams`

На первый взгляд стандарт `Reactive Streams` не выглядит сложным, но в действительности он содержит множество подводных камней. Кроме интерфейсов `Java` стандарт определяет множество правил реализации, которые представляют основную сложность. Правила накладывают строгие ограничения на каждый интерфейс, и очень важно обеспечить соответствие поведения реализации правилам, перечисленным в стандарте, что позволит в дальнейшем интегрировать реализации разных производителей без особых проблем. Это самый важный аспект, ради которого были сформулированы данные правила. К сожалению, на создание комплекта тестов, охватывающих все крайние случаи, может уйти намного больше времени, чем на правильную реализацию интерфейсов. С другой стороны, разработчикам нужен универсальный инструмент, с помощью которого они могли бы подтвердить соответствие стандарту своих реактивных библиотек и совместимость друг с другом. К счастью, такой набор тестов уже реализован Конрадом Малавски (Konrad Malawski). Он получил название `Reactive Streams Technology Compatibility Kit`, или просто `TCK`.



Узнать больше о TCK можно по ссылке <https://github.com/reactive-streams/reactive-streams-jvm/tree/master/tck>.

TCK следует всем инструкциям Reactive Streams и проверяет реализации на соответствие правилам. По сути, TCK – это набор тестов TestNG, которые должны адаптироваться и подготавливаться для проверки конкретной реализации Publisher или Subscriber. TCK включает полный список тестовых классов, призванных охватить все правила, описанные в стандарте Reactive Streams. Фактически имена всех классов соответствуют определенным правилам. Например, вот один из тестов, который можно найти в `org.reactivestreams.tck.PublisherVerification`.

```
...
void
required_spec101_subscriptionRequestMustResultInTheCorrectNumberOfProducedElements()
throws Throwable {
    ...
    ManualSubscriber<T> sub = env.newManualSubscriber(pub);           // (1)
    try {
        sub.expectNone(..., pub));                                   // (2)
        sub.request(1);                                              //
        sub.nextElement(..., pub));                                  //
        sub.expectNone(..., pub));                                   //
        sub.request(1);                                              //
        sub.request(2);                                              //
        sub.nextElements(3, ..., pub));                              //
        sub.expectNone(..., pub));                                   //
    } finally {
        sub.cancel();                                                // (3)
    }
    ...
}
```

Пояснения к коду.

1. Подписка на события проверяемого издателя. Reactive Streams TCK предоставляет собственные тестовые классы для проверки конкретного поведения.
2. Объявление ожиданий. Как можно заметить, здесь выполняется проверка конкретного поведения данного издателя Publisher согласно правилу 1.01. В данном случае контролируется, не посылает ли издатель Publisher событий больше, чем запрошено подписчиком Subscriber.
3. Здесь производится отмена подписки. Вне зависимости от успеха тест закрывает открытый ресурс и завершает взаимодействия. Отмена подписки производится с использованием ManualSubscriber API.

Важность теста заключается в проверке основной гарантии, которая должна обеспечиваться любой реализацией Publisher. Кроме того, все тесты в PublisherVerification гарантируют некоторое соответствие данной реализации Publisher требованиям стандарта Reactive Streams. Слова *некоторое соответствие* означают, что невозможно проверить все правила в полном объеме. Примером может служить правило 3.04, которое гласит, что запрос не должен приводить к сложным вычислениям, которые нельзя более или менее осмысленно проверить.

Проверка издателя Publisher

Необходимо не только осознать важность Reactive Streams TCK, но и знать, как пользоваться этим комплектом тестов. Чтобы получить некоторое представление об особенностях его работы, проверим один из компонентов нашей новой службы. Поскольку издатель Publisher является сердцем нашей системы, начнем с него. Как вы наверняка помните, для проверки основных аспектов поведения реализации Publisher TCK предлагает `org.reactivestreams.tck.PublisherVerification`. Вообще говоря, `PublisherVerification` – это абстрактный класс, который требует, чтобы мы переопределили всего два метода. Взгляните на следующий пример, чтобы понять, как организовать проверку созданного нами класса `NewsServicePublisher`.

```
public class NewsServicePublisherTest                                // (1)
    extends PublisherVerification<Newsletter> ... {                //

    public StreamPublisherTest() {                                  // (2)
        super(new TestEnvironment(...));                          //
    }                                                                //

    @Override                                                       // (3)
    public Publisher<Newsletter> createPublisher(long elements) { //
        ...                                                         //
        prepareItemsInDatabase(elements);                          // (3.1)
        Publisher<Newsletter> newsServicePublisher =              //
            new NewsServicePublisher(...);                          //
        ...                                                         //
        return newsServicePublisher;                                //
    }                                                                //

    @Override                                                       // (4)
    public Publisher<Newsletter> createFailedPublisher() {         //
        stopDatabase()                                              // (4.1)
        return new NewsServicePublisher(...);                      //
    }                                                                //
    ...                                                             //
}
```


Пояснения к коду.

1. Объявление класса `NewsServicePublisherTest`, наследующего класс `PublisherVerification`.
2. Объявление конструктора без параметров. Следует отметить, что `PublisherVerification` не имеет конструктора по умолчанию и требует от класса-потомка предоставить экземпляр `TestEnvironment`, отвечающий за подготовку настроек для теста, таких как тайм-ауты и журналы отладки.
3. Реализация метода `createPublisher`. Этот метод отвечает за создание экземпляра `Publisher`, который должен произвести указанное число элементов. В нашем случае, чтобы удовлетворить требованиям теста, необходимо заполнить базу данных определенным количеством новостей (3.1).
4. Реализация метода `createFailedPublisher`. Этот метод, в отличие от `createPublisher`, должен создать недействующий экземпляр `NewsServicePublisher`. Такой недействующий экземпляр `Publisher` можно получить, например, когда отсутствует доступ к источнику данных, что мы и обеспечиваем в строке (4.1).

Предыдущий тест расширяет базовую конфигурацию, необходимую для запуска проверки `NewsServicePublisher`. Предполагается, что реализация `Publisher` достаточно гибкая, чтобы сгенерировать заданное число элементов. Иначе говоря, тест может сообщить реализации `Publisher`, сколько элементов она должна сгенерировать и должен ли он работать нормально. С другой стороны, существует множество специфических случаев, когда `Publisher` должен сгенерировать только один элемент. Например, как вы наверняка помните, `NewsPreparationOperator` выводит только один элемент независимо от того, сколько получено им.

Просто определяя упомянутые настройки теста, нельзя проверить точность реализации `Publisher`, потому что многие тесты предполагают наличие более одного элемента в потоке. К счастью, Reactive Streams TCK учитывает такие случаи и позволяет настроить еще один метод с именем `maxElementsFromPublisher()`, который возвращает число, определяющее максимальное количество сгенерированных элементов.

```
@Override
public long maxElementsFromPublisher() {
    return 1;
}
```

С одной стороны, когда данный метод переопределяется, будут пропущены тесты, требующие более одного элемента. С другой – охват правил, определяемых стандартом Reactive Streams, сужается, и может потребоваться реализовать свои тестовые классы.

Проверка подписчика Subscriber

Упомянутые настройки – это минимум, необходимый для тестирования поведения производителя. Но кроме Publisher у нас еще есть экземпляры Subscriber, которые также необходимо протестировать. К счастью, это менее сложная группа правил в стандарте Reactive Stream, но их все равно требуется проверить, чтобы убедиться, что они соответствуют требованиям.

Проверить `NewsServiceSubscriber` можно с помощью двух комплектов тестов. Первый называется `org.reactivestreams.tck.SubscriberBlackboxVerification` и позволяет проверить реализацию Subscriber, не зная и/или не изменяя его внутреннее устройство. Комплект `SubscriberBlackboxVerification` удобно использовать, когда реализация Subscriber взята извне и нет никакого законного способа расширить ее поведение. Но при этом он охватывает ограниченное подмножество правил и не гарантирует полного соответствия проверяемой реализации. Чтобы увидеть, как можно проверить `NewsServiceSubscriber`, реализуем сначала проверку `SubscriberBlackboxVerification`.

```
public class NewsServiceSubscriberTest // (1)
    extends SubscriberBlackboxVerification<NewsLetter> { //
    public NewsServiceSubscriberTest() { // (2)
        super(new TestEnvironment()); //
    } //
    @Override // (3)
    public Subscriber<NewsLetter> createSubscriber() { //
        return new NewsServiceSubscriber(...); //
    } //
    @Override // (4)
    public NewsLetter createElement(int element) { //
        return new StubNewsLetter(element); //
    } //
    @Override // (5)
    public void triggerRequest(Subscriber<? super NewsLetter> s) { //
        ((NewsServiceSubscriber) s).eventuallyReadDigest(); // (5.1)
    } //
}
```

Пояснения к коду.

1. Определение класса `NewsServiceSubscriberTest`, который наследует тест `SubscriberBlackboxVerification`.
2. Объявление конструктора по умолчанию. По аналогии с `PublisherVerification` здесь мы создаем конкретный экземпляр `TestEnvironment`.

3. Реализация метода `createSubscriber`. В данном случае метод возвращает экземпляр `NewsServiceSubscriber`, который следует проверить на соответствие требованиям стандарта.
4. Реализация метода `createElement`. Метод играет роль фабрики новых элементов и генерирует новые экземпляры `NewsLetter`.
5. Реализация метода `triggerRequest`. Тест `SubscriberBlackboxVerification` предполагает недоступность внутренних механизмов подписчика, а значит, недоступность экземпляра подписки `Subscription` внутри `Subscriber`. То есть мы должны каким-то образом вручную запустить процесс, используя данный API (5.1).

В этом примере показан доступный API для проверки `Subscriber`. Кроме двух обязательных методов, `createSubscriber` и `createElement`, существует дополнительный метод для проверки внешнего метода `Subscription#request`. В нашем случае это полезное дополнение, позволяющее имитировать действия пользователя.

Второй комплект тестов называется `org.reactivestreams.tck.SubscriberWhiteboxVerification`. Он выполняет те же проверки, что и предыдущий комплект, но при этом, чтобы пройти тест, реализация `Subscriber` должна дополнительно предоставить возможность взаимодействия с `WhiteboxSubscriberProbe`.

```
public class NewsServiceSubscriberWhiteboxTest           // (1)
    extends SubscriberWhiteboxVerification<NewsLetter> { //
    ...                                                  //

    @Override                                           // (2)
    public Subscriber<NewsLetter> createSubscriber(     //
        WhiteboxSubscriberProbe<NewsLetter> probe      //
    ) {                                                 //
        return new NewsServiceSubscriber(...) {        //
            public void onSubscribe(Subscription s) {  //
                super.onSubscribe(s);                  // (2.1)
                probe.registerOnSubscribe(new SubscriberPuppet() { // (2.2)
                    public void triggerRequest(long elements) { //
                        s.request(elements);             //
                    }                                     //
                }
                public void signalCancel() {            //
                    s.cancel();                          //
                }                                       //
            }
        });                                           //
    }                                                 //
    public void onNext(NewsLetter newsLetter) {        //
        super.onNext(newsLetter);                      //
        probe.registerOnNext(newsLetter);              // (2.3)
    }                                                 //
```

```

        public void onError(Throwable t) {
            super.onError(t);
            probe.registerOnError(t);
        }
        public void onComplete() {
            super.onComplete();
            probe.registerOnComplete();
        }
    };
}
...
}

```

Пояснения к коду.

1. Объявление класса `NewsServiceSubscriberWhiteboxTest`, который наследует тест `SubscriberWhiteboxVerification`.
2. Реализация метода `createSubscriber`. Метод действует идентично одноименному методу в `SubscriberBlackboxVerification` и возвращает экземпляр `Subscriber`, но имеет дополнительный параметр `WhiteboxSubscriberProbe`, представляющий механизм, который встраивается для управления запросом и перехвата входящих сигналов. В отличие от теста `SubscriberBlackboxVerification`, надлежащей регистрацией обработчиков внутри `NewsServiceSubscriber` – (2.2), (2.3), (2.4) и (2.5) – тест способен не только посылать запросы, но и проверять, удовлетворены ли запросы и все ли элементы получены. Этот механизм управления более прозрачен, чем предыдущий. В строке (2.2) реализуется `SubscriberPuppet`, который поддерживает прямой доступ к полученной подписке `Subscription`.

Как видите, в отличие от `SubscriberBlackboxVerification`, тест `SubscriberWhiteboxVerification` требует расширения `Subscriber` и реализации дополнительных обработчиков. Тест `SubscriberWhiteboxVerification` охватывает более широкий круг правил, регламентирующих правильное поведение подписчика `Subscriber`, но не подходит для случаев, когда создается финальный класс, не позволяющий наследовать его.

Заключительный этап нашего погружения в вопросы проверки – тестирование реализации интерфейса `Processor`. Для этой цели в ТСК имеется `org.reactivestreams.tck.IdentityProcessorVerification`. Данный набор тестов способен проверить `Processor`, получающий и посылающий элементы одного типа. В нашем примере так действует только `martMulticastProcessor`. Поскольку тест должен проверить обе реализации, `Publisher` и `Subscriber`, `IdentityProcessorVerification` имеет такие же настройки, как у тестов для `Publisher` и `Subscriber`. По этой причине мы не будем погружаться в детали реализации теста, а рассмотрим только дополнительные методы, необходимые для проверки `SmartMulticastProcessor`.

```
public class SmartMulticastProcessorTest                // (1)
    extends IdentityProcessorVerification<NewsLetter> { //
    public SmartMulticastProcessorTest() {               // (2)
        super(..., 1);                                  //
    }                                                    //

    @Override                                           // (3)
    public Processor<Integer, Integer> createIdentityProcessor( //
        int bufferSize                                  //
    ) {                                                 //
        return new SmartMulticastProcessor<>();         //
    }                                                  //

    @Override                                           // (4)
    public NewsLetter createElement(int element) {      //
        return new StubNewsLetter(element);           //
    }                                                  //
}
```

Пояснения к коду.

1. Определение класса `SmartMulticastProcessorTest`, который наследует тест `IdentityProcessorVerification`.
2. Определение конструктора по умолчанию. Как можно заметить, кроме `TestEnvironment` (опущен в этом примере) мы передаем дополнительный параметр с количеством элементов, которое обработчик `Processor` должен кешировать. Мы знаем, что наша реализация `Processor` буферизует только один элемент, поэтому должны явно указать это число перед началом тестирования.
3. Реализация метода `createIdentityProcessor`, который возвращает проверяемый экземпляр `Processor`. Здесь `bufferSize` определяет количество элементов, которое обработчик `Processor` должен кешировать. Мы можем пропустить этот параметр, потому что размер внутреннего буфера уже был задан в конструкторе.
4. Реализация метода `createElement`. По аналогии с тестированием `Subscriber` мы должны определить фабричный метод для создания новых элементов.

В предыдущем примере показана основная конфигурация для тестирования `SmartMulticastProcessor`. Так как `IdentityProcessorVerification` наследует `SubscriberWhiteboxVerification` и `PublisherVerification`, общая конфигурация получается путем объединения конфигураций для каждого из них.

Итак, мы рассмотрели основной набор тестов, помогающих проверить поведение реактивных операторов. ТСК можно рассматривать как начальный набор интеграционных тестов. Однако вы должны иметь в виду, что наряду с проверкой с помощью ТСК каждый оператор должен тестироваться в отдельности.



Узнать больше о проверке с использованием TCK можно на странице проекта: <https://github.com/reactive-streams/reactive-streams-jvm/tree/master/tck>. Дополнительные примеры использования TCK имеются в репозитории Ratpack <https://github.com/ratpack/ratpack/tree/master/ratpack-exec/src/test/groovy/ratpack/stream/tck>. Существует также обширный список примеров использования TCK для проверки RxJava 2: <https://github.com/ReactiveX/RxJava/tree/2.x/src/test/java/io/reactivex/tck>.

JDK 9

Ценность стандарта была также замечена разработчиками JDK. Вскоре после публикации стандарта Даг Ли (Doug Lee) создал предложение о добавлении упомянутого интерфейса в JDK 9. Оно было обусловлено тем фактом, что текущая реализация Stream API предлагает только модель *PULL* (извлечения по запросу), а поддержка модели *PUSH* отсутствует. Вот что пишет Даг Ли (<http://jsr166-concurrency.10961.n7.nabble.com/jdk9-Candidate-classes-Flow-and-SubmissionPublisher-td11967.html>):

«...отсутствует единый гибкий асинхронный/параллельный API. Completable Future/CompletionStage в лучшем случае поддерживают программирование в стиле продолжений с использованием объектов Future, а java.util.Stream — (многоэтапные с возможностью распараллеливания) операции с элементами коллекций в стиле PULL. До настоящего момента единственной отсутствующей категорией остаются операции в стиле PUSH, которые выполняются активным источником по мере появления таких элементов».



Обратите внимание, что в Java Stream API используется *Splititerator* (модифицированная версия *Iterator*), поддерживающий возможность параллельного выполнения. Как помним, *Iterator* имеет метод *Iterator#next*, который реализует модель *PULL*, а не *PUSH*. Аналогично *Splititerator* имеет метод *tryAdvance*, являющийся комбинацией методов *hasNext* и *next* итератора *Iterator*. Отсюда можно сделать вывод, что в общем и целом Stream API основывается на модели *PULL*.

Главной целью предложения было добавление интерфейсов реактивных потоков в JDK. С целью его реализации все интерфейсы, определяемые стандартом Reactive Streams, добавили в класс `java.util.concurrent.Flow` в виде статических подклассов. С одной стороны, это значительное улучшение, потому что Reactive Streams становится также стандартной частью JDK. С другой стороны, многие производители уже опираются на спецификацию в пакете `org.reactivestreams.*`. А так как большинство производителей (таких как создатели RxJava) поддерживает несколько версий JDK, невозможно просто реализовать данные интерфейсы наряду с предыдущими. Соответственно, это улучшение вы-

двигает дополнительные требования к совместимости с JDK 9+ и к преобразованию одной спецификации в другую некоторым способом.

К счастью, стандарт Reactive Streams предлагает дополнительный модуль, который позволяет преобразовывать типы Reactive Streams в типы JDK Flow.

```
... // (1)
import org.reactivestreams.Publisher; //
import java.util.concurrent.Flow; //
... //
Flow.Publisher jdkPublisher = ...; // (2)
Publisher external = FlowAdapters.toPublisher(jdkPublisher) // (2.1)
Flow.Publisher jdkPublisher2 = FlowAdapters.toFlowPublisher( //
    external // (2.2)
); //
```

Пояснения к коду.

1. Судя по инструкциям `import`, мы должны импортировать `Publisher` из оригинальной библиотеки Reactive Streams и `Flow` – точку доступа ко всем интерфейсам реактивных потоков в JDK 9.
2. Определение экземпляра `Flow.Publisher` из JDK 9. В строке (2.1) используется метод `FlowAdapters.toPublisher` из оригинальной библиотеки Reactive Streams для преобразования `Flow.Publisher` в `org.reactivestreams.Publisher`. Для демонстрации этого в строке (2.2) выполняется обратное преобразование `org.reactivestreams.Publisher` в `Flow.Publisher` с использованием метода `FlowAdapters.toFlowPublisher`.

В предыдущем примере показано, как легко преобразовать `Flow.Publisher` в `org.reactivestreams.Publisher`. Следует отметить, что пример не связан с какой-либо действующей реализацией, потому что на момент создания этих строк отсутствовали хорошо известные реактивные библиотеки, написанные с нуля поверх JDK 9 Flow API, а значит, не было необходимости уходить от использования стандарта Reactive Streams как внешней библиотеки, поддерживающей JDK 6 и выше. Однако, скорее всего, в будущем многое изменится, обязательно появятся новые реактивные библиотеки, написанные в соответствии со стандартом Reactive Streams и перенесенные на JDK 9.



Отметим, что функции адаптера реализованы в виде отдельной библиотеки. Познакомиться со всеми доступными библиотеками можно по ссылке <http://www.reactive-streams.org/#jvm-interfaces-completed>.

Асинхронный и параллельный API в Reactive Streams

В предыдущих разделах мы обсудили основы поведения реактивных потоков Reactive Streams, однако в них ничего не говорилось об асинхронном и неблокирующем поведении реактивных конвейеров. Поэтому сейчас мы немного углубимся в стандарт Reactive Streams и исследуем эти черты.

С одной стороны, в правилах 2.2 и 3.4 Reactive Streams API утверждается, что обработка всех сигналов, посылаемых издателем Publisher и получаемых подписчиком Subscriber, должна производиться неблокирующим способом. Следовательно, мы можем быть уверены в эффективном использовании одного узла или одного ядра процессора в зависимости от среды выполнения.

С другой стороны, для эффективного использования всех процессоров или ядер необходима поддержка параллельных вычислений. Согласно стандарту Reactive Streams под параллельным выполнением подразумевается параллельный вызов метода `Subscriber#onNext`. К сожалению, правило 1.3 стандарта утверждает, что вызов методов `on***` должен осуществляться потокобезопасным способом и – если вызовы следуют из разных потоков выполнения – с использованием внешних механизмов синхронизации. Это означает, что все методы `on***` могут вызываться только последовательно. Также из этого следует, что нельзя написать, например, класс `ParallelPublisher` и обрабатывать элементы из потока параллельно.

Соответственно, возникает вопрос: как эффективно использовать ресурсы? Чтобы ответить на него, нужно проанализировать типичный конвейер обработки потока.

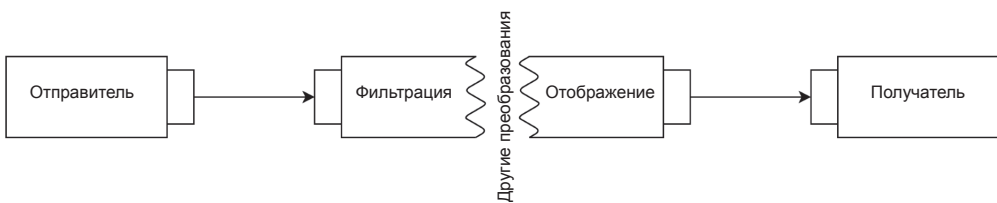


Рис. 3.10. Пример конвейера обработки потока с некоторой бизнес-логикой между отправителем и получателем

Как видим, в типичный конвейер помимо отправителя и получателя входит несколько этапов обработки или преобразования. Каждый этап может длиться продолжительное время и останавливать решение других задач.

Одно из решений в таких случаях – обмен асинхронными сообщениями между этапами. Для потоков данных, обрабатываемых в памяти, это означает, что часть обработки будет осуществляться в одном потоке выполнения Thread, а часть –

в другом. Например, потребление элементов на стороне получателя может быть очень тяжелой задачей с вычислительной точки зрения, которую рациональнее было бы выполнять в отдельном потоке выполнения Thread.



Рис. 3.11. Пример проведения асинхронной границы между отправителем и получателем

Распределяя обработку между двумя независимыми потоками выполнения, мы проводим асинхронную границу между этапами, тем самым распараллеливаем обработку элементов, потому что оба потока выполнения могут действовать независимо друг от друга. Чтобы добиться распараллеливания, необходимо использовать структуру данных, такую как очередь, чтобы надежно отделить этапы. Благодаря этому поток выполнения А сможет независимо поставлять элементы в очередь, а подписчик Subscriber в потоке выполнения В – извлекать их из той же очереди.

Распределение обработки между потоками выполнения приводит к дополнительным издержкам на обслуживание структуры данных. Однако это неизбежно из-за требований стандарта Reactive Streams. Число элементов в этой структуре данных обычно равно размеру пакета, который подписчик Subscriber запрашивает у издателя Publisher, но иногда может зависеть также от общего объема ресурсов, доступных системе.

В то же время перед разработчиками API встает вопрос: где в конвейере обработки должна проходить асинхронная граница? Здесь может быть как минимум три варианта. Первый – когда обработка осуществляется в одном потоке выполнения с отправителем (рис. 3.11). В этом случае все данные обрабатываются синхронно и по очереди преобразуются перед отправкой в другой поток выполнения. Второй вариант, полностью противоположный первому, – когда обработка осуществляется в одном потоке выполнения с получателем. Он может использоваться, когда создание новых элементов является ресурсоемкой задачей.

Третий вариант имеет место, когда обе задачи – создания и потребления элементов – являются ресурсоемкими. Тут наиболее эффективно вынести промежуточные преобразования в отдельный (третий) поток выполнения.

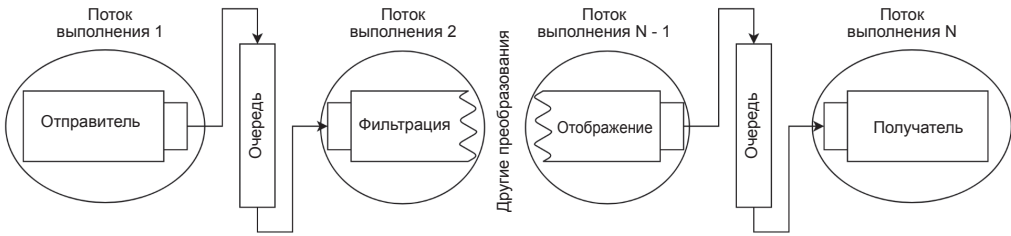


Рис. 3.12. Пример проведения асинхронных границ между всеми компонентами конвейера

Как показано на рис. 3.12, каждый этап обработки можно выделить в самостоятельный поток выполнения. Есть много способов настроить обработку потока данных, и каждый хорош при определенных условиях. Например, первый способ (рис. 3.11) идеально подходит для случаев, когда отправитель работает с меньшей нагрузкой, чем получатель, и операции по преобразованию выгоднее выполнять в рамках отправителя. И наоборот, когда получатель расходует меньше ресурсов, чем отправитель, логичнее обрабатывать данные в рамках получателя. Кроме того, иногда наиболее ресурсоемкой операцией оказывается само преобразование. В таком случае лучше отделить его от отправителя и получателя.

В любом случае важно помнить, что разделение обработки по разным потокам выполнения влечет за собой определенные накладные расходы, поэтому нужно найти правильный баланс между рациональным потреблением ресурсов (потоками выполнения и структурами данных) и эффективностью обработки элементов. Поиск такого баланса является еще одной задачей, которую сложно решить без удобного API.

К счастью, многие реактивные библиотеки, в том числе RxJava и Project Reactor, предлагают такой API. Мы не будем обсуждать имеющиеся возможности прямо сейчас, а вернемся к ним в главе 4 «*Project Reactor – основа реактивных приложений*».

Преобразование реактивного ландшафта

Факт поддержки стандарта Reactive Streams усилил значимость JDK 9, и это начало менять всю сферу. Лидеры в индустрии открытого программного обеспечения (такие как Netflix, Red Hat, Lightbend, MongoDB, Amazon и др.) стали внедрять это замечательное решение в свои продукты.

Изменения в RxJava

RxJava предлагает дополнительный модуль, позволяющий преобразовывать одни реактивные типы в другие. Например, посмотрим, как преобразовать `Observable<T>` в `Publisher<T>` и `rx.Subscriber<T>` в `org.reactivestreams.Subscriber<T>`.

Допустим, у нас есть приложение, использующее RxJava 1.x и тип `Observable` как главный механизм взаимодействий между компонентами.

```
interface LogService {  
    Observable<String> stream();  
}
```

Однако после публикации стандарта Reactive Streams было решено последовать за ним и абстрагировать наши интерфейсы от этой конкретной зависимости.

```
interface LogService {  
    Publisher<String> stream();  
}
```

Мы легко можем заменить тип `Observable` на `Publisher`. Но в процессе рефакторинга нам может потребоваться сделать нечто большее, чем просто заменить возвращаемый тип. К счастью, мы легко можем адаптировать существующую реализацию с `Observable` к `Publisher`.

```
class RxLogService implements LogService { // (1)  
    final HttpClient<...> rxClient = HttpClient.newClient(...); // (1.1)  
  
    @Override  
    public Publisher<String> stream() {  
        Observable<String> rxStream = rxClient.createGet("/logs") // (2)  
            .flatMap(...) //  
            .map(Utils::toString); //  
  
        return RxReactiveStreams.toPublisher(rxStream); // (3)  
    }  
}
```

Пояснения к коду.

1. Объявление класса `RxLogService`. Он представляет старую реализацию на основе Rx. В строке (1.1) используется `HttpClient` из `RxNetty`, который поддерживает асинхронные взаимодействия с внешними службами неблокирующим способом, применяя для `Netty Client`, завернутый в API на основе `RxJava`.
2. Отправка внешнего запроса. Здесь с помощью созданного экземпляра `HttpClient` внешней службе посылается запрос на получение потока записей из журнала, элементы которого преобразуются в экземпляры `String`.
3. С помощью библиотеки `RxReactiveStreams` экземпляр `rxStream` преобразуется в `Publisher`.

Как видите, разработчики RxJava позаботились о нас с вами и реализовали дополнительный класс `RxReactiveStreams`, способный преобразовать `Observable` в `Publisher` из Reactive Streams. Кроме того, с появлением стандарта Reactive

Streams разработчики RxJava реализовали нестандартизованную поддержку обратного давления, обеспечивающую совместимость преобразованного экземпляра Observable со стандартом Reactive Streams.

Кроме преобразования Observable в Publisher имеется возможность преобразовать rx.Subscriber в org.reactivestreams.Subscriber. Например, журнальные записи, о которых шла речь выше, раньше были сохранены в файле. С этой целью использовался экземпляр Subscriber, отвечавший за взаимодействия. В данном случае привести код в соответствие со стандартом Reactive Streams можно примерно так:

```
class RxFileService implements FileService {                                // (1)
    @Override                                                                // (2)
    public void writeTo(                                                    //
        String file,                                                        //
        Publisher<String> content                                           //
    ) {                                                                      //
        AsyncFileSubscriber rxSubscriber =                                 // (3)
            new AsyncFileSubscriber(file);                                  //
        content                                                             // (4)
            .subscribe(RxReactiveStreams.toSubscriber(rxSubscriber));      //
    }
}
```

Пояснения к коду.

1. Объявление класса RxFileService.
2. Реализация метода writeTo, который принимает Publisher как основной тип для взаимодействий между компонентами.
3. Объявление экземпляра AsyncFileSubscriber из RxJava.
4. Подписка на content. Чтобы беспрепятственно использовать экземпляр Subscriber из RxJava, он преобразуется с помощью того же вспомогательного класса RxReactiveStreams.

Как показано в примере, RxReactiveStreams предлагает обширный список методов для преобразования RxJava API в Reactive Streams API.

Аналогично любой экземпляр Publisher<T> можно преобразовать обратно в RxJava Observable.

```
Publisher<String> publisher = ...

RxReactiveStreams.toObservable(publisher).subscribe();
```

Проще говоря, библиотека RxJava начала следовать стандарту Reactive Streams. К сожалению, из-за необходимости поддерживать обратную совместимость реализация стандарта невозможна, и в будущем не планируется расширять стандарт

Reactive Streams для RxJava 1.x. Кроме того, с 31 марта 2018 года планируется прекратить поддержку RxJava 1.x.

К счастью, благодаря второму выпуску RxJava появились новые надежды. Давид Карнок (Dávid Karnok), автор второй версии библиотеки, значительно улучшил ее общий дизайн и ввел дополнительный тип, соответствующий стандарту Reactive Streams. Наряду с типом `Observable`, который остается неизменным для сохранения обратной совместимости, RxJava 2 предлагает новый реактивный тип с именем `Flowable`.

Тип `Flowable` имеет API, идентичный типу `Observable`, но при этом расширяет `org.reactivestreams.Publisher`. Как показано в следующем примере, экземпляр `Flowable` можно преобразовать в любые распространенные типы RxJava и обратно в тип, совместимый с Reactive Streams.

```
Flowable.just(1, 2, 3)
    .map(String::valueOf)
    .toObservable()
    .toFlowable(BackpressureStrategy.ERROR)
    .subscribe();
```

Как видите, для преобразования `Flowable` в `Observable` достаточно применить всего один оператор. Однако, чтобы выполнить обратное преобразование, `Observable` в `Flowable`, нужно реализовать некоторую стратегию обратного давления. В RxJava 2 тип `Observable` проектировался как поток, поддерживающий исключительно стратегию *PUSH*. Поэтому очень важно поддерживать совместимость преобразуемого `Observable` со стандартом Reactive Streams.



`BackpressureStrategy` – это стратегия поведения, когда производитель не учитывает потребностей потребителя. Иначе говоря, `BackpressureStrategy` определяет поведение потока данных с быстрым производителем и медленным потребителем. В начале главы мы обсуждали подобные ситуации и рассмотрели три основные стратегии, в том числе неограниченную буферизацию элементов, их удаление при переполнении и блокировку производителя при отсутствии спроса со стороны потребителя. Проще говоря, `BackpressureStrategy` отражает все описанные стратегии некоторым способом, кроме стратегии блокировки производителя, а также дополнительные стратегии, такие как `BackpressureStrategy.ERROR`, которая при отсутствии спроса посылает ошибку потребителю и автоматически отключает его. Мы не будем подробно рассматривать каждую стратегию в этой главе, а вернемся к ним в главе 4 «*Project Reactor – основа реактивных приложений*».

Изменения в Vert.x

Создатели остальных реактивных библиотек и фреймворков реализовали поддержку не только RxJava, но и стандарта Reactive Streams. Следуя требованиям спецификации, разработчики Vert.x включили дополнительный модуль, обеспечивающий поддержку Reactive Streams API. Следующий пример демонстрирует работу этого дополнения.

```
... // (1)
.requestHandler(request -> { //
    ReactiveReadStream<Buffer> rrs = // (2)
        ReactiveReadStream.readStream(); //
    HttpServletResponse response = request.response(); //

    Flowable<Buffer> logs = Flowable // (3)
        .fromPublisher(logsService.stream()) //
        .map(Buffer::buffer) //
        .doOnTerminate(response::end); //

    logs.subscribe(rrs); // (4)
    response.setStatusCode(200); // (5)
    response.setChunked(true); //
    response.putHeader("Content-Type", "text/plain"); //
    response.putHeader("Connection", "keep-alive"); //

    Pump.pump(rrs, response) // (6)
        .start(); //
})
...
```

Пояснения к коду.

1. Объявление обработчика запроса. Это обобщенный обработчик, способный обрабатывать любые запросы, отправленные серверу.
2. Объявление подписчика Subscriber и HTTP-ответа. Здесь ReactiveReadStream реализует оба интерфейса – org.reactivestreams.Subscriber и ReadStream, что позволяет преобразовать любую реализацию Publisher в источник данных, совместимый с Vert.x API.
3. Объявление потока обработки. В этом примере мы ссылаемся на новый интерфейс LogsService, основанный на Reactive Streams, и для преобразования элементов из потока данных используем Flowable API и RxJava 2.x.
4. Этап оформления подписки. После объявления потока обработки можно подписать ReactiveReadStream на Flowable.
5. Этап подготовки ответа.

6. Окончательный ответ отправляется клиенту. Класс `Pump` играет важную роль в сложном механизме управления обратным давлением для предотвращения переполнения буфера `WriteStream`.

Как видите, `Vert.x` не предлагает гибкого API для обработки потоков элементов, зато предоставляет API, позволяющий преобразовать любую реализацию `Publisher` в `Vert.x` API, сохранив поддержку управления обратным давлением согласно требованиям `Reactive Streams`.

Усовершенствования в `Ratpack`

Другой хорошо известный веб-фреймворк – `Ratpack` – также предлагает поддержку `Reactive Streams`, но в отличие от `Vert.x` – прямую поддержку. Например, вот как выглядит отправка записей в журнал, реализованная средствами `Ratpack`.

```
RatpackServer.start(server ->                                // (1)
  server.handlers(chain ->                                    //
    chain.all(ctx -> {                                         //
      Publisher<String> logs = logService.stream();           // (2)

      ServerSentEvents events = serverSentEvents(             // (3)
        logs,                                                  //
        event -> event.id(Objects.toString)                   // (3.1)
          .event("log")                                         //
          .data(Function.identity())                           //
      );                                                       //
      ctx.render(events);                                       // (4)
    })
  )
);
```

Пояснения к коду.

1. Начальное действие сервера и объявление обработчика запроса.
2. Объявление потока записей для журналирования.
3. Подготовка `ServerSentEvents`. Здесь упомянутый класс используется на этапе отображения для преобразования элементов, присылаемых издателем `Publisher`, в представление `ServerSentEvents`. Как видим, `ServerSentEvents` требует объявления функции отображения, описывающей, как отобразить элемент в набор полей `Event`.
4. Отображение потока данных.

Как показано в примере, поддержка `Reactive Streams` реализована в ядре `Ratpack`. Теперь один и тот же метод `LogService#stream` можно повторно использовать без необходимости дополнительного преобразования типов и без использования

дополнительных модулей для добавления поддержки конкретной реактивной библиотеки.

Кроме того, в отличие от библиотеки Vert.x, предлагающей упрощенную поддержку Reactive Streams, библиотека Ratpack включает собственную реализацию интерфейсов, определяемых стандартом. Эта функциональность доступна в классе `ratpack.stream.Streams`, программный интерфейс которого напоминает программный интерфейс RxJava.

```
Publisher<String> logs = logsService.stream();
TransformablePublisher publisher = Streams
    .transformable(logs)
    .filter(this::filterUsersSensitiveLogs)
    .map(this::escape);
```

Здесь Ratpack предлагает статическую фабрику для преобразования любой реализации `Publisher` в `TransformablePublisher`, которая, в свою очередь, дает возможность гибко обрабатывать потоки событий, используя знакомые операторы и этапы преобразования.

Драйвер MongoDB с поддержкой Reactive Streams

В предыдущих разделах мы рассмотрели поддержку Reactive Streams с точки зрения реактивных библиотек и фреймворков. Однако область применения стандарта не ограничивается фреймворками или реактивными библиотеками. Те же правила взаимодействий между производителем и потребителем можно применить к соединению с базой данных через драйвер этой базы данных.

Помимо обычных драйверов с интерфейсом обратных вызовов и на основе RxJava 1.x база данных MongoDB предлагает драйвер с поддержкой Reactive Streams и дополнительную реализацию Fluent API, поддерживающие механизм запросов с преобразованиями. Например, внутреннюю реализацию `DBPublisher`, которую мы видели в примере службы новостей, потенциально можно написать так, как показано ниже.

```
public class DBPublisher implements Publisher<News> {           // (1)
    private final MongoCollection<News> collection;           //
    private final Date publishedOnFrom;                       //

    public DBPublisher(                                       // (2)
        MongoClient client,
        Date publishedOnFrom
    ) { ... } //
    //

    @Override                                               // (3)
```



```
public void subscribe(Subscriber<? super News> s) {           //
    FindPublisher<News> findPublisher =                       // (3.1)
        collection.find(News.class);                          //
                                                            //
    findPublisher                                             // (3.2)
        .filter(Filters.and(                                  //
            Filters.eq("category", query.getCategory()),      //
            Filters.gt("publishedOn", today())                 //
        )                                                      //
        .sort(Sorts.descending("publishedOn"))               //
        .subscribe(s);                                         // (3.3)
}                                                            //
```

Пояснения к коду.

1. Объявление класса DBPublisher и его полей. Поле publishedOnFrom определяет начальную дату для поиска новостей.
2. Объявление конструктора. Один из параметров конструктора DBPublisher – настроенный клиент MongoDB типа com.mongodb.reactivestreams.client.MongoClient.
3. Реализация метода Publisher#subscriber. Здесь мы упростили реализацию из DBPublisher, используя FindPublisher из драйвера Reactive Streams MongoDB (3.1) и подписавшись на данного подписчика Subscriber (3.3). Как видим, FindPublisher предлагает гибкий API, позволяющий создавать выполняемые запросы с использованием функционального стиля программирования.

Наряду с поддержкой стандарта Reactive Streams драйвер MongoDB предлагает упрощенные средства для обработки запросов. Мы не будем вдаваться в тонкости реализации и поведения этого драйвера здесь, а вернемся к нему в главе 7 «Реактивный доступ к базам данных».

Комбинирование реактивных технологий на практике

Чтобы узнать больше о совместимости технологий, попробуем объединить несколько реактивных библиотек в приложении на основе Spring Framework 4. Оно будет опираться на последнюю версию службы новостей и предлагать клиентам для взаимодействий с ним простую конечную точку REST. Эта конечная точка осуществляет поиск новостей в базе данных и во внешних службах.

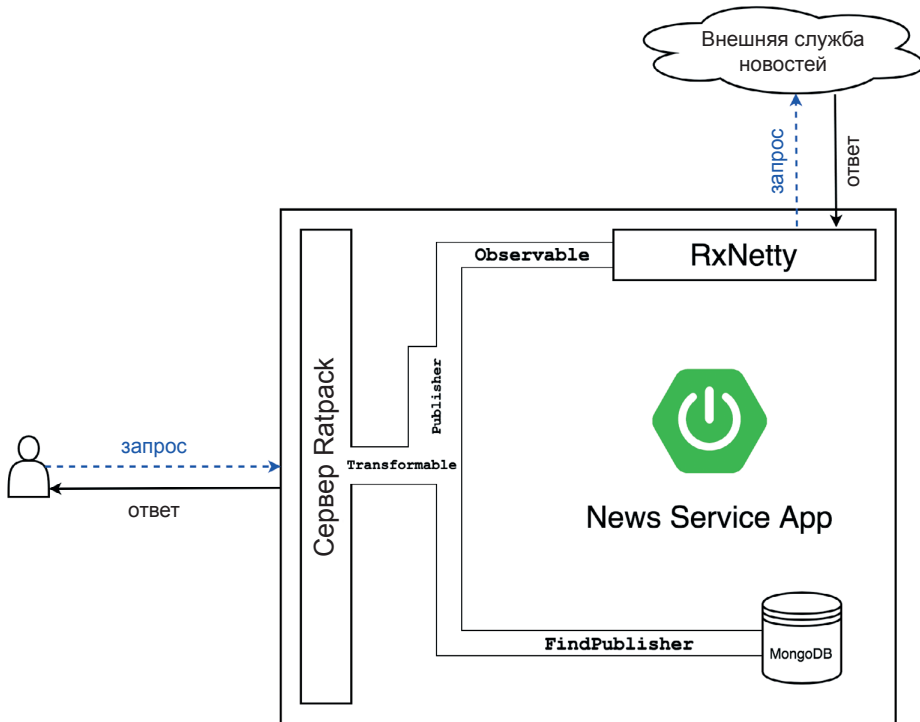


Рис. 3.13. Пример организации взаимодействий между библиотеками внутри общего приложения

Как показано на рис. 3.13, приложение использует три реактивные библиотеки. В качестве веб-сервера мы используем Ratpack. Класс `TransformablePublisher` из этой библиотеки позволяет легко объединять и обрабатывать результаты из нескольких источников. Одним из таких источников является база данных MongoDB, которая в ответ на запрос возвращает результат в форме `FindPublisher`. Наконец, мы обращаемся к внешней службе новостей и извлекаем данные с помощью HTTP-клиента RxNetty, который возвращает `Observable`, далее преобразуемый в `org.reactivestreams.Publisher`.

Итак, в нашем приложении имеется четыре компонента, первый из которых – Spring Framework 4, второй – Retrofit, играющий роль веб-фреймворка. Наконец, третий и четвертый компоненты – RxNetty и MongoDB – обеспечивают доступ к новостям. Мы не будем обсуждать особенности этих компонентов, отвечающих за связь с внешними службами, а рассмотрим только реализацию конечной точки. Это подчеркнет важность Reactive Streams как стандарта, помогающего объединить независимые библиотеки и фреймворки.

```
@SpringBootApplication                                // (1)
@EnableRatpack                                         // (1.1)
public class NewsServiceApp {                          //
```

```

@Bean // (2)
MongoClient mongoClient(MongoProperties properties) { ... } // (2.1)
@Bean //
DatabaseNewsService databaseNews() { ... } // (2.2)
@Bean //
HttpNewsService externalNews() { ... } // (2.3)

@Bean // (3)
public Action<Chain> home() { //
    return chain -> chain.get(ctx -> { // (3.1)
        FindPublisher<News> databasePublisher = // (4)
            databaseNews().lookupNews(); //
        Observable<News> httpNewsObservable = //
            externalNews().retrieveNews(); //
        TransformablePublisher<News> stream = Streams.merge( // (4.1)
            databasePublisher, //
            RxReactiveStreams.toPublisher(httpNewsObservable) //
        ); //
        ctx.render( // (5)
            stream.toList() //
                .map(Jackson::json) // (5.1)
        ); //
    }) //
} //

public static void main(String[] args) { // (6)
    SpringApplication.run(NewsServiceApp.class, args); //
} //
}

```

Пояснения к коду.

1. Объявление класса `NewsServiceApp`. Он снабжен аннотацией `@SpringBootApplication`, которая предполагает применение Spring Boot. В строке (1.1) используется дополнительная аннотация `@EnableRatpack`, являющаяся частью модуля `ratpack-spring-boot` и обеспечивающая автоматическую настройку сервера Ratpack.
2. Обычное объявление компонентов JavaBeans. В строке (2.1) выполняется настройка компонента `MongoClient`, в (2.2) и (2.3) – настройка служб для поиска и извлечения новостей.
3. Объявление обработчика запросов. Чтобы создать обработчик запросов Ratpack, мы объявили компонент в (3.1) с типом `Action<Chain>`, который позволит определить конфигурацию обработчика.
4. Вызов служб и объединение результатов. Здесь мы вызываем методы служб и объединяем возвращаемые потоки данных с помощью Ratpack Streams API (4.1).

5. Этап отображения объединенного потока данных. Все элементы асинхронно заворачиваются в список, который затем преобразуется в конкретное отображение, например JSON (5.1).
6. Реализация метода `main`. Используется типичный способ запуска приложений Spring Boot.

Предыдущий пример на практике показывает возможности стандарта Reactive Streams. С помощью API нескольких независимых библиотек мы легко смогли сконструировать единый поток и вернуть результат конечному пользователю, не испытав особых сложностей при объединении нескольких реактивных библиотек. Единственное исключение из правила – класс `HttpNewsService`, метод `retrieveNews` которого возвращает `Observable`. Однако, как вы помните, `RxReactiveStreams` предлагает целый список вспомогательных методов, помогающих преобразовать `Observable` из `RxJava 1.x` в `Publisher`.

Заключение

Как показал предыдущий пример, Reactive Streams существенно улучшает совместимость реактивных библиотек. Мы также узнали, что лучший способ проверить совместимость `Publisher` – использовать набор тестов `Technology Compatibility Kit`, который следует всем положениям стандарта Reactive Streams.

В то же время стандарт определяет гибридную модель взаимодействий *PULL-PUSH*. Это дополнение решает проблему управления обратным давлением и одновременно повышает гибкость, предлагая возможность выбора модели для использования.

После добавления поддержки стандарта Reactive Streams в JDK9 его значимость резко возросла. Однако появились некоторые издержки, связанные с необходимостью преобразования типов между двумя вариантами спецификации.

Стандарт Reactive Streams предлагает несколько способов организации взаимодействий между операторами. Такая гибкость дает возможность проводить границы асинхронности любым удобным способом, однако накладывает большую ответственность на разработчиков реактивных библиотек, потому что такие решения должны соответствовать потребностям бизнеса. Кроме того, решения должны быть достаточно гибкими, чтобы их можно было настраивать со стороны API.

Изменяя поведение реактивных потоков данных, стандарт также меняет реактивный ландшафт. Лидеры индустрии открытого программного обеспечения, такие как Netflix, Redhead, Lightbend, Pivotal и др., внедрили поддержку стандарта в свои продукты. Однако для пользователей Spring Framework наиболее значительным изменением в реактивном мире стало появление новой библиотеки – **Project Reactor**.

Значение Project Reactor трудно переоценить, поскольку она легла в основу новой реактивной экосистемы Spring. Следовательно, прежде чем углубляться в исследование внутренней реализации этой экосистемы, мы должны поближе познакомиться с Project Reactor и оценить ее важность. В следующей главе рассмотрим концептуальные компоненты Project Reactor и пример их использования.

Глава 4

Project Reactor – основа реактивных приложений

В предыдущей главе мы познакомились со стандартом Reactive Streams и с тем, как он способствовал унификации реактивных библиотек, предложив общие интерфейсы и новую модель *PULL-PUSH* обмена данными.

В этой главе исследуем Project Reactor, самую известную библиотеку в реактивном ландшафте, которая уже стала важнейшей частью экосистемы Spring Framework. Рассмотрим наиболее важные и широко используемые Project Reactor API. Эта библиотека настолько универсальна и многофункциональна, что заслуживает отдельной книги, поэтому невозможно описать ее API во всех подробностях в одной главе. Тем не менее мы заглянем внутрь этой библиотеки и попробуем создать реактивное приложение с ее использованием.

Будут рассмотрены следующие темы:

- история и мотивация создания Project Reactor;
- терминология и API библиотеки Project Reactor API;
- продвинутые возможности Project Reactor;
- наиболее важные детали реализации Project Reactor;
- сравнение часто используемых реактивных типов;
- пример реализации приложения с использованием библиотеки Reactor.

Краткая история Project Reactor

Как рассказано в предыдущей главе, стандарт Reactive Streams обеспечивает совместимость реактивных библиотек, а также решает проблему управления обратным давлением, определяя модель *PULL-PUSH* обмена данными. Несмотря на существенные улучшения, стандарт Reactive Streams все же является всего лишь стандартом – он только определяет API и правила и не предлагает никаких фак-

тических библиотек. В этой главе мы рассмотрим одну из самых популярных реализаций стандарта Reactive Streams – библиотеку Project Reactor (или просто Reactor). Библиотека Reactor стремительно развивалась с момента ее основания, и в настоящее время это самая современная реактивная библиотека. Давайте познакомимся с ее историей, чтобы увидеть, как стандарт Reactive Streams повлиял на формирование API и детали реализации библиотеки.

Project Reactor 1.x

При подготовке стандарта Reactive Streams разработчикам из Spring Framework потребовался высокопроизводительный фреймворк обработки данных. Особая надобность в нем возникла в проекте Spring XD. Он был нужен, чтобы упростить разработку приложений, действующих с большими данными. Чтобы удовлетворить эту потребность, команда Spring запустила новый проект. С самого начала он предусматривал поддержку асинхронных, неблокирующих операций. Команда назвала его Project Reactor. По сути, версия Reactor 1.x включала самые передовые приемы обработки сообщений, такие как Reactor Pattern, и поддерживала функциональный и реактивный стили программирования.



Reactor Pattern – это шаблон поведения, помогающий реализовать асинхронную и поочередную обработку событий. Это означает, что все события ставятся в очередь и обрабатываются позже в отдельном потоке выполнения. События пересылаются всем подписчикам (обработчикам событий) и обрабатываются поочередно. Дополнительную информацию о Reactor Pattern можно найти по ссылке <http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf>.

Благодаря этим приемам версия Project Reactor 1.x давала возможность писать компактный код.

```
Environment env = new Environment(); // (1)
Reactor reactor = Reactors.reactor() // (2)
    .env(env) //
    .dispatcher(Environment.RING_BUFFER) // (2.1)
    .get(); //

reactor.on($"channel", // (3)
    event -> System.out.println(event.getData())); //

Executors.newSingleThreadScheduledExecutor() // (4)
    .scheduleAtFixedRate( //
        () -> reactor.notify("channel", Event.wrap("test")), //
        0, 100, TimeUnit.MILLISECONDS //
    ); //
```

Пояснения к коду.

1. Создается экземпляр `Environment`. Он определяет контекст выполнения, который используется при создании экземпляра `Dispatcher`. Потенциально может возвращать разные типы диспетчеров – от действующих внутри процесса до распределенных.
2. Создается экземпляр `Reactor` – прямая реализация `Reactor Pattern`. В предыдущем примере используется класс `Reactors` – гибкий построитель конкретных экземпляров `Reactor`. В строке (2.1) используется предопределенная реализация `Dispatcher`, основанная на структуре `RingBuffer`. Узнать больше об особенностях и общем дизайне `Dispatcher` на основе `RingBuffer` можно по ссылке <https://martinfowler.com/articles/lmax.html>.
3. Объявление селектора канала и потребителя событий. Здесь мы регистрируем обработчика событий (в данном случае лямбда-выражение, которое выводит все полученные события в `System.out`). Фильтрация событий производится с помощью строкового селектора, который определяет имя канала события. Селекторы `Selectors`.`$` предлагают широкий выбор критериев, вследствие чего окончательное выражение селектора может оказаться очень сложным.
4. Настройка производителя событий `Event` в форме планируемого задания, для чего используются возможности `ScheduledExecutorService`. В результате получается задание, выполняющееся периодически и посылающее события `Event` в конкретный канал, созданный экземпляром `Reactor`.

За кулисами события обслуживаются диспетчером `Dispatcher` и пересылаются в точку назначения. В зависимости от реализации `Dispatcher` события могут обслуживаться синхронно или асинхронно. Такой подход обеспечивает функциональную декомпозицию и в целом напоминает обработку событий в `Spring Framework`. Кроме того, `Reactor 1.x` предлагает множество удобных оберток, позволяющих с легкостью конструировать конвейеры обработки событий.

```

... // (1)
Stream<String> stream = Streams.on(reactor, $("channel")); // (2)
stream.map(s -> "Hello world " + s) // (3)
    .distinct() //
    .filter((Predicate<String>) s -> s.length() > 2) //
    .consume(System.out::println); // (3.1)

Deferred<String, Stream<String>> input = Streams.defer(env); // (4)

Stream<String> compose = input.compose() // (5)
compose.map(m -> m + " Hello World") // (6)
    .filter(m -> m.contains("1")) //
    .map(Event::wrap) //
    .consume(reactor.prepare("channel")); // (6.1)

```



```
for (int i = 0; i < 1000; i++) { // (7)
    input.accept(UUID.randomUUID().toString()); //
} //
```

Пояснения к коду.

1. Создаются экземпляры `Environment` и `Reactor`, как в предыдущем примере.
2. Создается экземпляр `Stream`, позволяющий организовывать цепочки преобразований в функциональном стиле. Применяв метод `Streams.on` к экземпляру `Reactor` с указанным селектором, получаем объект `Stream`, связанный с указанным каналом в данном экземпляре `Reactor`.
3. Определяется поток обработки. Мы выполняем несколько промежуточных операций, таких как `map`, `filter` и `consume`. Последняя из них играет роль завершающего оператора (3.1).
4. Создается отложенный поток данных. Класс `Deferred` – это специальная обертка, которая позволяет передавать в поток `Stream` события, созданные вручную. В данном случае метод `Stream.defer` создает дополнительный экземпляр класса `Reactor`.
5. Создается экземпляр `Stream`. Для этого мы извлекаем `Stream` из экземпляра `Deferred` вызовом метода `compose`.
6. Создается реактивный поток обработки событий. Эта часть конвейера аналогична той, что определена в строке (3). В строке (6.1) используем типичный шаблон кода `Reactor API`: `e -> reactor.notify("channel", e)`.
7. Генерируется случайный элемент и передается экземпляру `Deferred`.

В предыдущем примере мы подписывались на канал, а затем последовательно обрабатывали все входящие события. Кроме того, в нем используется декларативный подход к определению конвейера обработки. Мы определяем два отдельных этапа. Также обратите внимание, что код выглядит как хорошо известный `RxJava API`, что делает его более привычным для пользователей `RxJava`. Благодаря этому `Reactor 1.x` хорошо интегрировался с `Spring Framework`. Кроме средств обработки сообщений `Reactor 1.x` предлагает множество расширений, например для `Netty`.

Подводя итог, можно сказать, что `Reactor 1.x` прекрасно справлялся с задачей быстрой обработки событий. Благодаря превосходной интеграции с `Spring Framework` и поддержке `Netty` эта библиотека позволяла писать высокопроизводительные системы, поддерживающие асинхронную и неблокирующую обработку сообщений.

Однако версия `Reactor 1.x` имела также некоторые недостатки. Прежде всего библиотека не поддерживала возможности управления обратным давлением. К сожалению, реализация `Reactor 1.x` позволяла управлять обратным давлением только через блокировку потока выполнения производителя или удаление событий. Кроме того, обработка ошибок была очень сложна в реализации.

Reactor 1.x предлагает несколько разных вариантов обработки ошибок и отказов. Несмотря на все недостатки, библиотека Reactor 1.x была использована в популярном веб-фреймворке Grails. И конечно же, она существенно повлияла на следующую версию реактивной библиотеки.

Project Reactor 2.x

Вскоре после выпуска официальной версии Reactor 1.x в специальную группу Reactive Streams Special Interest Group был приглашен Стивен Малдини (Stephane Maldini), эксперт по системам высокопроизводительной обработки сообщений и один из руководителей проекта Project Reactor. Эта группа, как нетрудно догадаться, работала над стандартом Reactive Streams. Обретя новое понимание природы Reactive Streams и передав знания команде Reactor, в начале 2015 года Стивен Малдини и Джон Брисбин (Jon Brisbin) анонсировали выход Reactor 2.x. Тогда Стивен Малдини заявил: *«Reactor 2 стала первым выстрелом в Reactive Streams»*.

Самым значительным изменением в дизайне Reactor стало выделение EventBus и Stream в самостоятельные модули. Кроме того, глубокое перепроектирование новой версии библиотеки Reactor Streams обеспечило ее полное соответствие стандарту Reactive Streams. Команда Reactor значительно усовершенствовала API библиотеки. Например, новый Reactor API лучше интегрируется с Java Collections API.

Вторая версия Reactor Streams API стала больше похожей на RxJava API. Наряду с простыми дополнениями для создания и потребления потоков данных появились средства управления обратным давлением, потоками выполнения и устойчивостью, как показано ниже.

```
stream
    .retry()                                // (1)
    .onOverflowBuffer()                     // (2)
    .onOverflowDrop()                       //
    .dispatchOn(new RingBufferDispatcher("test")) // (3)
```

Предыдущий пример демонстрирует три простых приема.

1. Однострочным оператором `retry` мы увеличиваем устойчивость процесса обработки, то есть когда в потоке выше возникает ошибка, операции в нем повторяются.
2. Вызовы методов `onOverflowBuffer` и `onOverflowDrop` добавляют управление обратным давлением для случая, когда издатель поддерживает только модель *PUSH* (и не учитывает спрос потребителя).

3. Вызовом оператора `dispatchOn` мы выделили нового диспетчера `Dispatcher` для работы с этим реактивным потоком данных. Таким образом обеспечивается возможность асинхронной обработки сообщений.

Реализация `Reactor EventBus` также была усовершенствована. Прежде всего объект `Reactor`, отвечающий за отправку сообщений, переименовали в `EventBus`. Модуль также перепроектировали, добавили поддержку стандарта `Reactive Streams`.

Примерно в то же время Стивен Малдини встретился с Дэвидом Карноком (David Karnok), который активно работал над диссертацией «*High resolution and transparent production informatics*». В диссертации освещались исследования в области реактивных потоков данных, реактивного программирования и `RxJava`. Стивен Малдини и Дэвид Карнок объединили свои идеи и опыт разработки `RxJava` и `Project Reactor` и воплотили их в библиотеке `reactive-stream-commons`. Позднее она легла в основу `Reactor 2.5` и, наконец, `Reactor 3.x`.



Исходный код библиотеки `reactive-streams-commons` доступен в репозитории GitHub: <https://github.com/reactor/reactive-streams-commons>.

После года напряженной работы вышла версия `Reactor 3.0`. В то же время появилась почти идентичная ей версия `RxJava 2.0`. Последняя имеет больше сходства с `Reactor 3.x`, чем ее предшественница `RxJava 1.x`. Самое большое отличие между этими двумя библиотеками заключается в том, что `RxJava` нацелена на `Java 6` (включая поддержку `Android`), тогда как за основу `Reactor 3` была выбрана версия `Java 8`. Кроме того, `Reactor 3.x` определила реактивные метаморфозы в `Spring Framework 5`. Именно поэтому `Project Reactor` широко используется в остальных главах книги.

А теперь познакомимся с программным интерфейсом `Project Reactor 3.x` и рассмотрим эффективные приемы его использования.

Основы Project Reactor

С самого начала библиотека `Reactor` задумывалась с целью избавить разработчиков от **ада обратных вызовов** и **глубоко вложенного кода** при создании асинхронных конвейеров. Мы описали эти явления и осложнения, которые они вызывают, в главе 1 «*Причины выбора Spring*». В поисках методов, помогающих сделать код более линейным, авторы библиотеки сформулировали аналогию сборочной линии: «*Процесс обработки данных в реактивном приложении можно представить как сборочную линию. Reactor является одновременно и конвейерной лентой, и постами сборки*».

Главная цель библиотеки – увеличить **удобочитаемость** кода и обеспечить возможность **компоновки** рабочих процессов, определяя их с помощью инструментов из библиотеки `Reactor`. Общедоступный API проектировался как очень вы-

сокоуровневый и вместе с тем универсальный, но в то же время он не жертвует производительностью. Программный интерфейс предлагает богатый выбор операторов (*«постов на сборочной линии»*), обеспечивающих получение большей добавочной стоимости по сравнению с «чистым» стандартом Reactive Streams.

Reactor API позволяет составлять цепочки из операторов, благодаря чему можно конструировать сложные графы выполнения, потенциально пригодные для многократного использования. Отметим, что такой граф выполнения лишь определяет процесс обработки, который ничего не делает, пока подписчик фактически не оформит подписку, то есть только **подписка запускает поток фактической обработки данных**.

Библиотека проектировалась для достижения **максимальной эффективности при манипулировании данными** – как локальными, так и извлекаемыми асинхронными запросами с возможными сбоями в операциях ввода/вывода. Именно по этой причине операторы обработки ошибок в Project Reactor такие гибкие и, как увидим далее, помогают писать **отказоустойчивый** код.

Как мы уже знаем, **управление обратным давлением** является важнейшим свойством, которому уделяется особое внимание в стандарте Reactive Streams. А поскольку Reactor реализует этот стандарт, управление обратным давлением является одной из центральных задач данной библиотеки. Следовательно, когда заходит разговор о реактивных потоках данных, реализованных средствами Reactor, подразумевается, что данные перетекают от издателя к подписчику, а в обратном направлении, от подписчика к издателю, распространяются сигналы управления, как показано на рис. 4.1.



Рис. 4.1. Распространение данных и сигналов управления в реактивном потоке

Библиотека поддерживает распространение обратного давления для всех возможных моделей:

- **PUSH:** подписчик запрашивает фактически бесконечное количество элементов вызовом `subscription.request(Long.MAX_VALUE)`;
- **PULL:** подписчик запрашивает следующий элемент только после обработки предыдущего: `subscription.request(1)`;
- **PULL-PUSH** (иногда эту модель называют **смешанной**): подписчик управляет передачей элементов в режиме реального времени, а издатель подстраивается под заявленную скорость потребления данных.

Кроме того, для адаптации под старый API, не поддерживающий модель *PULL-PUSH*, Reactor предлагает массу *классических* механизмов управления обратным давлением, а именно кеширование, кадрирование, удаление избыточных сообщений, возбуждение исключений и т. д. Все эти методы мы рассмотрим ниже в данной главе. В некоторых случаях упомянутые стратегии позволяют организовать предварительную выборку данных еще до того, как появится фактический спрос, что помогает повысить отзывчивость системы. Также Reactor API – достаточно эффективный инструмент, чтобы сгладить короткие пики пользовательской активности и предотвратить перегрузку системы.

Библиотека Project Reactor проектировалась так, чтобы не зависеть от конкретного механизма параллельного выполнения, поэтому в ней не используются никакие модели параллельного выполнения. В то же время она предлагает набор планировщиков для управления потоками выполнения практически любым способом, и если ни один из предлагаемых планировщиков не соответствует имеющимся требованиям, то разработчик может реализовать своего планировщика с полноценным низкоуровневым управлением. Тема управления потоками выполнения в библиотеке Reactor также рассматривается далее.

А теперь, после краткого знакомства с библиотекой Reactor, создадим новый проект и приступим к исследованию ее богатого API.

Добавление библиотеки Reactor в проект

Предполагаем, что читатель уже знаком со стандартом Reactive Streams. Если это не так, прочитайте предыдущую главу. В текущем контексте знание стандарта Reactive Streams очень важно, потому что Project Reactor построена поверх него. Единственной обязательной зависимостью библиотеки Project Reactor является пакет `org.reactivestreams:reactive-streams`.

Чтобы подключить Project Reactor к проекту приложения, достаточно добавить следующую зависимость в файл `build.gradle`.

```
compile("io.projectreactor:reactor-core:3.2.0.RELEASE")
```

На момент написания книги самой свежей была версия библиотеки `3.2.0.RELEASE`. Она используется и в Spring Framework 5.1.



Процедура добавления библиотеки Project Reactor в проект Maven описана в статье https://projectreactor.io/docs/core/3.2.0.RELEASE/reference/#_maven_installation.

Также имеет смысл добавить в проект следующую зависимость, предоставляющую необходимый набор инструментов для тестирования реактивного кода, который, как вы понимаете, мы должны охватить модульными тестами.

```
testCompile("io.projectreactor:reactor-test:3.2.0.RELEASE")
```

В этой главе мы используем лишь несколько простых приемов тестирования реактивных потоков, а более подробно вопросы тестирования реактивного кода рассмотрим в главе 9 «Тестирование реактивных приложений».

Теперь, добавив Reactor в путь поиска классов (classpath), мы готовы к экспериментам с реактивными типами и операторами из Reactor.

Реактивные типы: Flux и Mono

Как мы уже знаем, стандарт Reactive Streams определяет всего четыре интерфейса: `Publisher<T>`, `Subscriber<T>`, `Subscription` и `Processor<T, R>`. Постараемся следовать именно в таком порядке и рассмотрим реализации интерфейсов, предлагаемые библиотекой.

Прежде всего Project Reactor предлагает две реализации интерфейса `Publisher<T>`: `Flux<T>` и `Mono<T>`. Это добавляет реактивным типам дополнительный контекстный смысл. В процессе изучения поведения реактивных типов (`Flux` и `Mono`) мы используем некоторые реактивные операторы, но не будем подробно рассказывать, как они действуют. Подробнее операторы будут обсуждаться далее.

Flux

Диаграмма на рис. 4.2 иллюстрирует, как протекает поток данных через класс `Flux`.

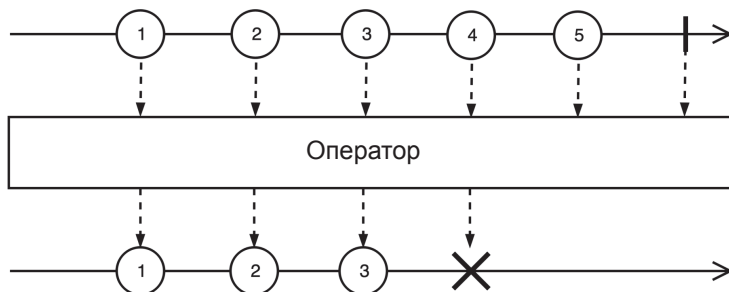


Рис. 4.2. Пример преобразования одного потока данных Flux в другой

Класс Flux определяет самый обычный реактивный поток данных, который может вернуть ноль, один или несколько элементов и даже бесконечное число элементов. Он имеет следующую формулу:

```
onNext x 0..N [onError | onComplete]
```

В императивном мире редко приходится работать с бесконечными контейнерами данных, но они широко распространены в функциональном программировании. Следующий код создает простой бесконечный реактивный поток:

```
Flux.range(1, 5).repeat()
```

Этот поток снова и снова производит числа от 1 до 5 (последовательность 1, 2, 3, 4, 5, 1, 2, ...). Он не представляет никакой проблемы, не переполнит память, потому что для получения каждого элемента не требуется заранее генерировать всю последовательность целиком. Кроме того, подписчик в любой момент может отменить подписку и превратить бесконечный поток в конечный.

Будьте осторожны: попытка собрать все элементы, производимые бесконечным потоком, может вызвать `OutOfMemoryException`. Поэтому поступать так в промышленных приложениях не рекомендуется. Воспроизвести это нежелательное поведение можно с помощью следующего кода:

```
Flux.range(1, 100)                                // (1)
    .repeat()                                       // (2)
    .collectList()                                 // (3)
    .block();                                       // (4)
```

Пояснения к коду.

1. Оператор `range` создает последовательность целых чисел от 1 до 100 (включительно).
2. Оператор `repeat` повторно подписывается на получение реактивного потока после его исчерпания. То есть оператор `repeat` подписывается на поток, принимает элементы от 1 до 100 и сигнал `onComplete` и затем снова подписывается на поток, опять принимает элементы от 1 до 100 и т. д.
3. С помощью оператора `collectList` пытаемся собрать все произведенные элементы в один большой список. Конечно, из-за того что оператор `repeat` генерирует бесконечный поток, список постоянно увеличивается в размерах, пока наконец не заполнит всю доступную память и не вызовет сбой приложения с ошибкой `java.lang.OutOfMemoryError: Java heap space`. Наше приложение просто исчерпает всю свободную память.
4. Оператор `block` создает фактическую подписку и блокирует поток выполнения до получения окончательного результата, который в данном случае так и не будет получен, потому что поток данных не имеет конца.

Mono

Теперь посмотрим, чем тип Mono отличается от типа Flux (см. рис. 4.3).

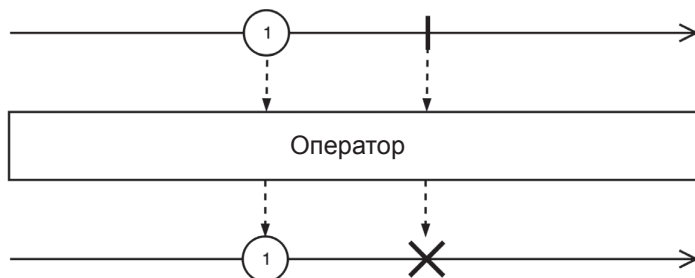


Рис. 4.3. Пример преобразования одного потока данных Mono в другой

В отличие от Flux, тип Mono определяет поток данных, который может произвести **не более одного элемента** и описывается следующей формулой:

```
onNext x 0..1 [onError | onComplete]
```

Различия между Flux и Mono не только позволяют придавать дополнительный смысл сигнатурам методов, но и обеспечивают более эффективную реализацию Mono благодаря отсутствию промежуточных буферов и необходимости выполнять дорогостоящую синхронизацию.

Тип Mono<T> пригодится в том случае, если API приложения может вернуть не более одного элемента. Как следствие он легко заменит CompletableFuture<T>, предложив похожую семантику. Конечно, эти два типа имеют некоторые небольшие семантические различия – CompletableFuture, в отличие от Mono, не может завершиться нормально без передачи значения. Кроме того, CompletableFuture начинает обработку немедленно, тогда как Mono ничего не делает до создания подписки. Преимущество типа Mono заключается в поддержке большого количества реактивных операторов и возможности безупречной интеграции в большой реактивный конвейер.

Также Mono можно использовать для уведомления клиента о завершении действия. В таких случаях можно вернуть тип Mono<Void> и послать сигнал onComplete() по завершении обработки или onError() в случае ошибки. В подобном сценарии никакие данные не возвращаются. Возвращается только сигнал, который, кстати, можно использовать для запуска других вычислений.

Экземпляры типов Mono и Flux без труда можно «преобразовывать» друг в друга. Например, Flux<T>.collectList() вернет Mono<List<T>>, а Mono<T>.flux() – Flux<T>. Плюс библиотека автоматически оптимизирует некоторые преобразования, не изменяющие семантику. Например, взгляните на следующее преобразование (Mono -> Flux -> Mono):


```
Mono.from(Flux.from(mono))
```

Предыдущий код вернет оригинальный экземпляр `mono`, потому что концептуально этот код представляет собой пустое преобразование.

Реактивные типы из RxJava 2

Несмотря на то что библиотеки RxJava 2.x и Project Reactor имеют одну и ту же основу, RxJava 2 предлагает иной набор реактивных издателей. Так как эти библиотеки реализуют одни и те же идеи, имеет смысл рассмотреть отличия RxJava 2, по крайней мере касающиеся реактивных типов. Все остальные аспекты, включая операторы, управление потоками выполнения и обработку ошибок, очень похожи. То есть после знакомства с одной библиотекой можно считать, что познакомились с обеими.

Как рассказывалось в главе 2 «*Реактивное программирование в Spring. Основные понятия*», первоначально библиотека RxJava 1.x имела только один реактивный тип: `Observable`. Позднее в нее были добавлены типы `Single` и `Completable`. Во второй версии в библиотеке появились следующие реактивные типы: `Observable`, `Flowable`, `Single`, `Maybe` и `Completable`. Давайте посмотрим, чем они отличаются и чем похожи на тандем типов `Flux`/`Mono`.

Observable

Тип `Observable` в RxJava 2 предлагает почти ту же семантику, что и в RxJava 1.x, но больше не принимает значений `null`. Кроме того, `Observable` не поддерживает управление обратным давлением и не реализует интерфейс `Publisher`. То есть он не совместим напрямую со стандартом `Reactive Streams`. Поэтому будьте внимательны, используя его в потоках с большим количеством элементов (больше пары тысяч). С другой стороны, тип `Observable` имеет меньше накладных расходов, чем тип `Flowable`. У него есть метод `toFlowable`, преобразующий поток в `Flowable` применением стратегии обратного давления по выбору пользователя.

Flowable

Тип `Flowable` – прямой аналог типа `Flux` из библиотеки `Reactor`. Он реализует стандартный интерфейс `Publisher`. Как следствие его можно использовать в реактивных конвейерах, реализованных с помощью `Project Reactor`, потому что хорошо продуманный API будет потреблять аргументы типа `Publisher` вместо более специализированного типа `Flux`.

Single

Тип `Single` представляет поток данных, генерирующий точно один элемент. Он не наследует интерфейс `Publisher`, а также имеет метод `toFlowable`, который не требует указывать стратегию обратного давления. Тип `Single` лучше представля-

ет семантику `CompletableFuture`, тип `Mono` из библиотеки `Reactor`. При всем при этом он не начинает обработку до создания подписки.

Maybe

Чтобы предложить ту же семантику, что и тип `Mono` из `Reactor`, библиотека `RxJava 2.x` реализует тип `Maybe`. Однако он несовместим со стандартом `Reactive Streams`, потому что не реализует интерфейс `Publisher`. Для этого в нем имеется метод `toFlowable`.

Completable

Также в `RxJava 2.x` есть тип `Completable`, способный генерировать только сигналы `onError` и `onComplete`, но не поддерживающий сигнал `onNext`. Он реализует интерфейс `Publisher` и имеет метод `toFlowable`. Семантически этот тип соответствует типу `Mono<Void>`, который не может сгенерировать сигнал `onNext`.

Подводя итог, отметим, что в `RxJava 2` более тонкие семантические различия между реактивными типами. Только тип `Flowable` совместим со стандартом `Reactive Streams`. Тип `Observable` решает ту же задачу, но не имеет поддержки обратного давления. Тип `Maybe<T>` соответствует типу `Mono<T>` в `Reactor`, а тип `Completable` – типу `Mono<Void>`. Семантику типа `Single` нельзя напрямую воспроизвести в терминах `Project Reactor`, потому что ни один из типов не гарантирует минимального числа событий. Для интеграции с другим кодом, совместимым со стандартом `Reactive Streams`, типы из `RxJava` следует преобразовать в тип `Flowable`.

Создание последовательностей Flux и Mono

`Flux` и `Mono` имеют множество фабричных методов для создания реактивных потоков на основе уже имеющихся данных. Например, можно создать экземпляр `Flux` из ссылок на объекты или из коллекции и даже свой «ленивый» диапазон чисел.

```
Flux<String> stream1 = Flux.just("Hello", "world");
Flux<Integer> stream2 = Flux.fromArray(new Integer[]{1, 2, 3});
Flux<Integer> stream3 = Flux.fromIterable(Arrays.asList(9, 8, 7));
```

Как показано ниже, метод `range` позволяет сгенерировать поток целых чисел, где 2010 – начальное значение, а 9 – число элементов в последовательности.

```
Flux<Integer> stream4 = Flux.range(2010, 9);
```

Это очень удобный способ получить последовательность последних лет. Предыдущий код сгенерирует поток целых чисел.

2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018

`Mono` предлагает похожие фабричные методы, но ориентирован на единственный элемент в потоке. Он также часто используется в комбинации с типами `nullable` и `Optional`.

```
Mono<String> stream5 = Mono.just("One");
Mono<String> stream6 = Mono.justOrEmpty(null);
Mono<String> stream7 = Mono.justOrEmpty(Optional.empty());
```

`Mono` может очень пригодиться для обертывания асинхронных операций, таких как HTTP-запросы или запросы к базе данных. Для этого в `Mono` реализованы следующие методы: `fromCallable(Callable)`, `fromRunnable(Runnable)`, `fromSupplier(Supplier)`, `fromFuture(CompletableFuture)`, `fromCompletionStage(CompletionStage)` и др. Завернуть долго выполняющийся HTTP-запрос в тип `Mono` можно, например, так:

```
Mono<String> stream8 = Mono.fromCallable(() -> httpRequest());
```

Предыдущий код можно переписать иначе, если воспользоваться синтаксисом ссылок на методы в Java 8.

```
Mono<String> stream8 = Mono.fromCallable(this::httpRequest);
```

Обратите внимание, что предыдущий код не только выполняет HTTP-запрос асинхронно (обеспечивается соответствующим экземпляром `Scheduler`), но и обрабатывает ошибки, которые могут передаваться как сигнал `onError`.

Оба типа, `Flux` и `Mono`, позволяют адаптировать любой другой экземпляр `Publisher` с помощью фабричного метода `from(Publisher<T> p)`.

Оба реактивных типа имеют методы создания удобных и часто используемых на практике пустых потоков данных, а также потоков, содержащих только ошибки.

```
Flux<String> empty = Flux.empty();
Flux<String> never = Flux.never();
Mono<String> error = Mono.error(new RuntimeException("Unknown id"));
```

`Flux` и `Mono` имеют фабричный метод `empty()`, генерирующий пустые экземпляры `Flux` и `Mono` соответственно. Аналогично метод `never()` создает поток, который не пошлет ни одного сигнала – `onComplete`, `onNext` или `onError`.

Фабричный метод `error(Throwable)` создает последовательность, которая передает ошибку через вызов метода `onError(...)` каждому подписчику, когда тот подпишется. Ошибка создается при определении экземпляра `Flux` или `Mono`, и соответственно каждый подписчик получит один и тот же экземпляр `Throwable`.

Фабричный метод `defer` создает последовательность, которая определяет свое поведение в момент подписки, и соответственно может генерировать разные данные для разных подписчиков.

```
Mono<User> requestUserData(String sessionId) {  
    return Mono.defer(() ->  
        isValidSession(sessionId)  
            ? Mono.fromCallable(() -> requestUser(sessionId))  
            : Mono.error(new RuntimeException("Invalid user session")));  
}
```

Этот код откладывает проверку `sessionId` до появления фактической подписки. Напротив, следующий код выполнит проверку при вызове метода `requestUserData(...)`, который может произойти до или после фактической подписки (кроме того, подписка вообще может никогда не состояться).

```
Mono<User> requestUserData(String sessionId) {  
    return isValidSession(sessionId)  
        ? Mono.fromCallable(() -> requestUser(sessionId))  
        : Mono.error(new RuntimeException("Invalid user session"));  
}
```

Первый пример проверяет сеанс, когда кто-то подпишется на возвращаемый `Mono<User>`. Второй выполняет проверку, только когда вызывается метод `requestUserData`. Факт создания подписки не запускает проверку.

Подводя итог, скажем, что Project Reactor позволяет создавать последовательности Flux и Mono простым перечислением элементов в вызове метода `just`. Мы легко можем завернуть `Optional` в Mono вызовом `justOrEmpty` или `Supplier` в Mono – вызовом `fromSupplier`. Мы можем отобразить экземпляр `Future` с помощью метода `fromFuture` или `Runnable` – с помощью `fromRunnable`. Кроме того, можем преобразовать массив или коллекцию `Iterable` в поток Flux с помощью метода `fromArray` или `fromIterable`. Также Project Reactor дает возможность создавать более сложные реактивные последовательности, о которых рассказывается далее в этой главе. А теперь давайте посмотрим, как потреблять элементы, производимые реактивным потоком.

Подписка на реактивный поток

Как можно догадаться, Flux и Mono предлагают перегруженные лямбда-версии метода `subscribe()`, который упрощает процедуру подписки.

```
subscribe(); // (1)  
subscribe(Consumer<T> dataConsumer); // (2)  
subscribe(Consumer<T> dataConsumer, // (3)
```

```
Consumer<Throwable> errorConsumer);  
  
subscribe(Consumer<T> dataConsumer,                      // (4)  
          Consumer<Throwable> errorConsumer,  
          Runnable completeConsumer);  
  
subscribe(Consumer<T> dataConsumer,                      // (5)  
          Consumer<Throwable> errorConsumer,  
          Runnable completeConsumer,  
          Consumer<Subscription> subscriptionConsumer);  
  
subscribe(Subscriber<T> subscriber);                      // (6)
```

Давайте рассмотрим имеющиеся у нас варианты создания подписчиков. Прежде всего отметим, что все перегруженные версии метода `subscribe` возвращают экземпляр интерфейса `Disposable`. Это обстоятельство можно использовать для отмены подписки `Subscription`. В случаях (1)-(4) метод `subscription` запрашивает неограниченное число элементов (`Long.MAX_VALUE`).

А теперь рассмотрим различия.

1. Это самый простой способ подписаться на поток данных, так как метод игнорирует любые сигналы. Обычно предпочтительнее использовать другие варианты, однако иногда бывает полезно запустить потоковую обработку, имеющую побочные эффекты.
2. Для каждого значения вызывает `dataConsumer` (сигнал `onNext`). Эта версия не обрабатывает сигналы `onError` и `onComplete`.
3. Действует так же, как и версия (2), но позволяет обрабатывать сигнал `onError`. Сигнал `onComplete` игнорируется.
4. Действует так же, как и версия (3), но позволяет обрабатывать сигнал `onComplete`.
5. Позволяет получить все элементы из реактивного потока, в том числе сигналы об ошибках и сигнал завершения. Важно отметить, что эта версия дает возможность управлять подпиской путем запроса требуемого объема данных (конечно же, ничто не мешает запросить `Long.MAX_VALUE`).
6. Самый обобщенный способ подписки. Здесь можно передать свою реализацию подписчика `Subscriber` с желаемым поведением. Но, несмотря на универсальность этого варианта, он редко используется на практике.

Давайте создадим простой реактивный поток и подпишемся на него.

```
Flux.just("A", "B", "C")  
    .subscribe(  
        data -> log.info("onNext: {}", data),  
        err -> { /* игнорируется */ },  
        () -> log.info("onComplete"));
```

Предыдущий код выведет в консоль следующие строки:

```
onNext: A
onNext: B
onNext: C
onComplete
```

Стоит еще раз отметить, что простая неограниченная подписка (`Long.MAX_VALUE`) иногда может вынудить производителя выполнить значительный объем работы для удовлетворения потребителя. Поэтому если производитель больше подходит для обработки ограниченного спроса, рекомендуется организовать управление спросом с помощью объекта подписки или применяя операторы, ограничивающие требования, как будет показано далее в этой главе.

Подпишемся на реактивный поток, используя подписку с ручным управлением.

```
Flux.range(1, 100)                                //(1)
    .subscribe(                                     //(2)
        data -> log.info("onNext: {}", data),
        err -> { /* игнорируется */ },
        () -> log.info("onComplete"),
        subscription -> {                           //(3)
            subscription.request(4);                 //(3.1)
            subscription.cancel();                   //(3.2)
        }
    );
```

Этот код выполняет следующие операции.

1. Сначала с помощью оператора `range` генерируется 100 значений.
2. Выполняется подписка на поток, так же как это делалось в предыдущем примере.
3. Однако теперь мы управляем подпиской. Сначала запрашивается четыре элемента (3.1), а потом подписка сразу же отменяется (3.2), то есть другие элементы вообще не должны генерироваться.

Если запустить этот код, он выведет в консоль следующие строки:

```
onNext: 1
onNext: 2
onNext: 3
onNext: 4
```

Обратите внимание, что мы не получили сигнал `onComplete`, потому что подписчик отменил подписку до завершения потока данных. Также важно помнить, что реактивный поток может быть завершен производителем (с сигналом `onError` или `onComplete`) или подписчиком через экземпляр `Subscription`. Кроме того, для отмены подписки можно использовать экземпляр `Disposable`. Обычно его

применяет не подписчик, а код, находящийся на более высоком уровне абстракции. Например, отменим обработку потока, вызвав `Disposable`.

```
Disposable disposable = Flux.interval(Duration.ofMillis(50)) // (1)
    .subscribe( // (2)
        data -> log.info("onNext: {}", data)
    );
Thread.sleep(200); // (3)
disposable.dispose(); // (4)
```

Этот код выполняет следующие операции.

1. Фабричный метод `interval` позволяет генерировать события с заданной периодичностью (в данном случае каждые 50 мс). Генерируемый им поток данных не имеет конца.
2. Подписываясь на поток, мы задаем только обработчик сигналов `onNext`.
3. Ждем некоторое время, чтобы получить несколько событий ($200 : 50 = 4$, то есть должно быть сгенерировано примерно четыре события).
4. Вызов метода `dispose`, который отменяет подписку.

Реализация своих подписчиков

Если метод по умолчанию `subscribe(...)` не обеспечивает необходимой гибкости, можно реализовать своего собственного подписчика `Subscriber`. Мы всегда можем сами реализовать интерфейс `Subscriber`, определяемый стандартом `Reactive Streams`, и подписать его на поток данных.

```
Subscriber<String> subscriber = new Subscriber<String>() {
    volatile Subscription subscription; // (1)

    public void onSubscribe(Subscription s) { // (2)
        subscription = s; // (2.1)
        log.info("initial request for 1 element"); //
        subscription.request(1); // (2.2)
    }

    public void onNext(String s) { // (3)
        log.info("onNext: {}", s); //
        log.info("requesting 1 more element"); //
        subscription.request(1); // (3.1)
    }

    public void onComplete() {
        log.info("onComplete");
    }

    public void onError(Throwable t) {
```

```
        log.warn("onError: {}", t.getMessage());
    }
};

Flux<String> stream = Flux.just("Hello", "world", "!");           // (4)
stream.subscribe(subscriber);                                     // (5)
```

Поясним, что делает наша реализация Subscriber.

1. Наш подписчик должен хранить ссылку на экземпляр подписки Subscription, связывающий издателя Publisher и нашего подписчика Subscriber. Поскольку подписка и обработка данных могут происходить в разных потоках выполнения, мы использовали ключевое слово `volatile`, чтобы обеспечить правильность ссылки на экземпляр Subscription во всех потоках выполнения.
2. После оформления подписки наш подписчик Subscriber информируется обратным вызовом `onSubscribe`. В нем мы сохраняем подписку (2.1) и посылаем запрос с начальным требованием (2.2). Без такого запроса ТСК-совместимый производитель не будет посылать данные и обработка элементов вообще никогда не начнется.
3. В обратном вызове `onNext` регистрируем полученные данные и запрашиваем следующий элемент. В данном случае используем простую модель *PULL* (`subscription.request(1)`) управления обратным давлением.
4. Генерируем простой поток данных с помощью фабричного метода `just`.
5. Подписываем нашего подписчика на реактивный поток, который определен в строке (4).

Если запустить этот код, он выведет в консоль следующие строки:

```
initial request for 1 element
onNext: Hello
requesting 1 more element
onNext: world
requesting 1 more element
onNext: !
requesting 1 more element
onComplete
```

Однако здесь описан неправильный подход к определению подписки. Он рушит *линейность* кода и способствует появлению ошибок. Но самое сложное – мы должны организовать управление обратным давлением и обеспечить соответствие подписчика всем требованиям ТСК. Кроме того, в предыдущем примере мы нарушили несколько требований ТСК, касающихся проверки и отмены подписки.

Чтобы избежать таких проблем, рекомендуется наследовать класс `BaseSubscriber` из библиотеки Project Reactor. В данном случае более удачная версия подписчика могла бы выглядеть так:


```
class MySubscriber<T> extends BaseSubscriber<T> {
    public void hookOnSubscribe(Subscription subscription) {
        log.info("initial request for 1 element");
        request(1);
    }

    public void hookOnNext(T value) {
        log.info("onNext: {}", value);
        log.info("requesting 1 more element");
        request(1);
    }
}
```

Кроме методов `hookOnSubscribe(Subscription)` и `hookOnNext(T)` можно переопределить такие методы, как `hookOnError(Throwable)`, `hookOnCancel()`, `hookOnComplete()`, и др. Класс `BaseSubscriber` предлагает методы для более точного управления требованиями – `request(long)` и `requestUnbounded()`. Кроме того, с классом `BaseSubscriber` намного проще реализовать ТСК-совместимого подписчика. Такой подход выглядит более предпочтительным, когда сам подписчик обладает ценными ресурсами со своим жизненным циклом. Например, подписчик может включать обработчик файлов или соединение `WebSocket` со сторонней службой.

Преобразование реактивных последовательностей с помощью операторов

При работе с реактивными последовательностями, кроме создания и обработки потока данных, важно также иметь возможность преобразовывать их и манипулировать ими. Только тогда реактивное программирование превращается в полезный прием. Project Reactor предлагает инструменты (обычные и фабричные методы) практически для любых преобразований реактивной информации. В общем случае средства, предлагаемые библиотекой, можно разделить на следующие категории:

- преобразование имеющихся последовательностей;
- методы просмотра при обработке последовательностей;
- разделение и объединение последовательностей `Flux`;
- работа с временем;
- синхронный возврат данных.

Мы не можем описать здесь все операторы и фабричные методы, потому что для этого требуется очень много места и почти невозможно вспомнить их все. К тому же в этом нет большой необходимости, если учесть, что Project Reactor сопровождается превосходной документацией, в том числе руководством по вы-

бору подходящих операторов: <http://projectreactor.io/docs/core/release/reference/#which-operator>. Тем не менее в этом разделе мы расскажем о наиболее часто используемых операторах и используем несколько примеров кода.

Обратите внимание, что большинство операторов имеет множество вариантов, расширяющих базовое поведение. Кроме того, в каждой версии Project Reactor появляются новые полезные операторы. Поэтому обязательно обратитесь к документации Reactor, где вы всегда найдете самую свежую информацию об операторах.

Отображение элементов реактивных последовательностей

Самый естественный способ преобразования последовательности – отображение каждого элемента в некоторое новое значение. Типы `Flux` и `Mono` предлагают оператор `map`, который действует подобно оператору `map` из Java Stream API. Функция с сигнатурой `map(Function<T, R>)` позволяет обрабатывать элементы по одному. Конечно, из-за того что она меняет тип элемента с `T` на `R`, вся последовательность также меняет свой тип, то есть после оператора `map` тип `Flux<T>` превращается в тип `Flux<R>`, а `Mono<T>` – в `Mono<R>`. На рис. 4.4 показана диаграмма работы `Flux.map()`.

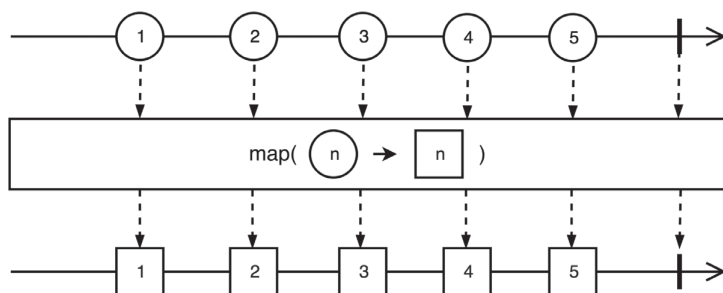


Рис. 4.4. Оператор `map`

Оператор `map` класса `Mono` действует похожим образом. Оператор `cast(Class c)` преобразует элементы из потока данных в целевой класс. Самый простой, пожалуй, способ реализовать оператор `cast(Class c)` – использовать оператор `map()`. Если заглянуть в исходный код класса `Flux`, там можно увидеть следующие строки, подтверждающие наше предположение.

```
public final <E> Flux<E> cast(Class<E> clazz) {
    return map(clazz::cast);
}
```

Оператор `index` позволяет присвоить индексы элементам последовательности. Он имеет следующую сигнатуру: `Flux<Tuple2<Long, T>> index()`. То есть те-

перь мы должны работать с классом `Tuple2`. Он представляет структуру данных *кортеж*, которая отсутствует в стандартной библиотеке Java. Библиотека Reactor предлагает классы от `Tuple2` до `Tuple8`, которые часто используются библиотечными операторами. Оператор `timestamp` действует подобно оператору `index`, но вместо индексов добавляет текущее время. Таким образом, следующий код должен присвоить элементам последовательности индексы и текущее время.

```
Flux.range(2018, 5)                                // (1)
    .timestamp()                                    // (2)
    .index()                                        // (3)
    .subscribe(e -> log.info("index: {}, ts: {}, value: {}",    // (4)
        e.getT1(),                                           // (4.1)
        Instant.ofEpochMilli(e.getT2().getT1()),           // (4.2)
        e.getT2().getT2()));                                // (4.3)
```

Поясним, что делает предыдущий код.

1. С помощью оператора `range` генерируются некоторые данные (числа от 2018 до 2022). Этот оператор возвращает последовательность типа `Flux<Integer>`.
2. С помощью оператора `timestamp` к значениям присоединяется текущее время. Теперь последовательность имеет тип `Flux<Tuple2<Long, Integer>>`.
3. К последовательности применяется оператор `index`. Теперь последовательность имеет тип `Flux<Tuple2<Long, Tuple2<Long, Integer>>>`.
4. Оформляется подписка на последовательность, и производится регистрация ее элементов в журнале. Вызов `e.getT1()` возвращает индекс (4.1), а вызов `e.getT2().getT1()` – метку времени, значение которой выводится в удобочитаемом формате с помощью класса `Instant` (4.2), а вызов `e.getT2().getT2()` возвращает фактическое значение (4.3).

Если выполнить предыдущий код, он выведет вот что:

```
index: 0, ts: 2018-09-24T03:00:52.041Z, value: 2018
index: 1, ts: 2018-09-24T03:00:52.061Z, value: 2019
index: 2, ts: 2018-09-24T03:00:52.061Z, value: 2020
index: 3, ts: 2018-09-24T03:00:52.061Z, value: 2021
index: 4, ts: 2018-09-24T03:00:52.062Z, value: 2022
```

Фильтрация реактивных последовательностей

Библиотека Project Reactor содержит также все, что необходимо для фильтрации элементов, в том числе:

- оператор `filter` пропускает только элементы, удовлетворяющие заданному условию;

- оператор `ignoreElements` возвращает `Mono<T>` и отфильтровывает все элементы. Получившаяся в результате последовательность завершается, только когда завершается оригинальная последовательность;
- предлагая оператор `take(n)`, библиотека помогает ограничить число получаемых элементов. Этот метод игнорирует все элементы, кроме n первых;
- `takeLast` возвращает только последний элемент из потока данных;
- `takeUntil(Predicate)` пропускает элементы, пока не выполнится заданное условие;
- `elementAt(n)` пропускает только n -й элемент;
- оператор `single` передает единственный элемент от источника и сигнализирует ошибкой `NoSuchElementException`, если источник пуст, и ошибкой `IndexOutOfBoundsException`, если источник передаст больше одного элемента.

Кроме того, поддерживается возможность перепрыгивать через элементы или извлекать их, ориентируясь не только на их количество, но и на продолжительность `Duration`, используя операторы `skip(Duration)` и `take(Duration)`.

Также перепрыгивать через элементы или извлекать их можно до момента получения некоего сообщения из другого потока данных, используя операторы `takeUntilOther(Publisher)` и `skipUntilOther(Publisher)`.

Рассмотрим пример, когда мы должны начать обрабатывать данные и прекратить обработку после получения некоего события из другого потока данных. Вот как это можно организовать:

```
Mono<?> startCommand = ...
Mono<?> stopCommand = ...
Flux<UserEvent> streamOfData = ...

streamOfData
    .skipUntilOther(startCommand)
    .takeUntilOther(stopCommand)
    .subscribe(System.out::println);
```

В данном случае мы можем начать и прекратить обработку элементов, но только один раз. На рис. 4.5 показана временная диаграмма процесса.

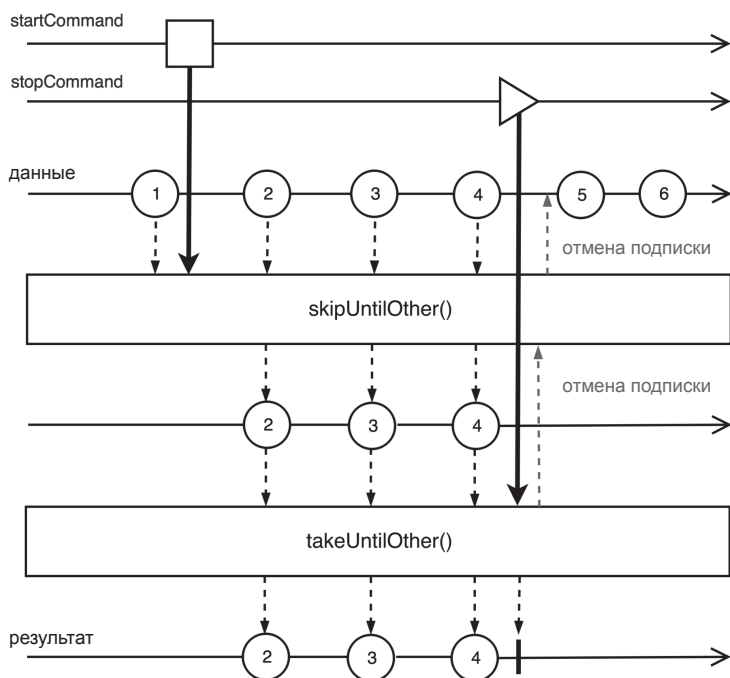


Рис. 4.5. Выборка элементов из потока между командами startCommand/stopCommand

Сбор данных из реактивных последовательностей

С помощью `Flux.collectList()` и `Flux.collectSortedList()` можно собрать все элементы в списке и обработать результат как поток `Mono`. Второй метод не только собирает данные, но и сортирует их. Взгляните на следующий пример.

```
Flux.just(1, 6, 2, 8, 3, 1, 5, 1)
    .collectSortedList(Comparator.reverseOrder())
    .subscribe(System.out::println);
```

Он выведет элементы в виде единой сортированной коллекции.

```
[8, 6, 5, 3, 2, 1, 1, 1]
```



Отметим, что такой способ накопления элементов последовательности в коллекции может оказаться ресурсоемкой затеей, особенно если последовательность состоит из большого количества элементов. Кроме того, подобный подход может привести к ошибке исчерпания доступной памяти, если попытаться собрать данные из бесконечного потока.

Project Reactor позволяет собрать элементы из Flux не только в коллекции `List`, но и в коллекциях других типов:

- в коллекции `Map` (`Map<K, T>`) с помощью оператора `collectMap`;
- в коллекции `Map<K, Collection<T>>` с помощью оператора `collectMultiMap`;
- в любой структуре данных, реализующей интерфейс `java.util.stream.Collectors`, с помощью оператора `Flux.collect(Collector)`.

Оба типа, `Flux` и `Mono`, имеют методы `repeat()` и `repeat(times)`, позволяющие организовать цикл по входящим последовательностям. Мы уже использовали их в предыдущем разделе.

Еще один удобный метод – `defaultIfEmpty(T)` – позволяет подставлять значения по умолчанию в пустые `Flux` и `Mono`.

`Flux.distinct()` пропускает только элементы, не встречавшиеся в потоке прежде. Однако этот метод хранит все уникальные элементы, поэтому использовать его следует с большой осторожностью, особенно при обработке потоков данных большой мощности. Метод `distinct` имеет варианты, позволяющие подставлять собственные алгоритмы определения повторяющихся элементов. Благодаря этому иногда можно оптимизировать использование ресурсов оператором `distinct`.



Под *большой мощностью* понимается большое количество необычных или уникальных элементов. Например, потоки с идентификационными номерами или именами пользователей обычно обладают большой мощностью. В то же время значения перечислений или значения из небольших словарей фиксированного размера не обладают большой мощностью.

Оператор `Flux.distinctUntilChanged()` не страдает таким ограничением и может использоваться с бесконечными потоками для устранения дубликатов, следующих друг за другом. Это поведение демонстрирует диаграмма на рис. 4.6.

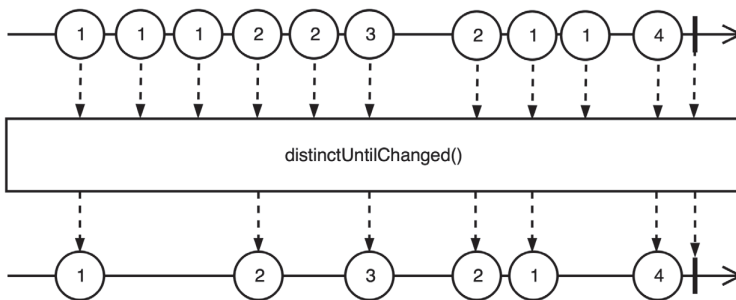


Рис. 4.6. Оператор `distinctUntilChanged`

Сокращение элементов потока

Project Reactor позволяет подсчитать число элементов в потоке с помощью `count()` или убедиться, что все элементы обладают требуемым свойством, с по-

мощью `Flux.all(Predicate)`. Также с помощью оператора `Flux.any(Predicate)` легко можно проверить наличие в потоке хотя бы одного элемента с требуемыми свойствами.

С помощью оператора `hasElements` можно проверить наличие в потоке хотя бы одного любого элемента, а с помощью оператора `hasElement` – наличие элемента с требуемыми свойствами. Последний реализует логику вычислений по короткой схеме и завершается со значением `true`, как только обнаруживает элемент с соответствующим значением. Оператор `any` позволяет проверить не только значение элемента, но и любое другое его свойство, если передать ему соответствующий экземпляр `Predicate`. Например, проверим наличие в потоке четного числа.

```
Flux.just(3, 5, 7, 9, 11, 15, 16, 17)
    .any(e -> e % 2 == 0)
    .subscribe(hasEvens -> log.info("Has evens: {}", hasEvens));
```

Оператор `sort` сортирует элементы в фоновом режиме и посылает отсортированную последовательность, как только оригинальная завершится.

Класс `Flux` позволяет также осуществить свертку (folding) последовательности с использованием нестандартной логики. Оператору `reduce` обычно передается начальное значение и функция, которая объединит предыдущее значение со следующим. Например, подсчитаем сумму целых чисел от 1 до 10.

```
Flux.range(1, 5)
    .reduce(0, (acc, elem) -> acc + elem)
    .subscribe(result -> log.info("Result: {}", result));
```

В результате должно получиться число 15. Оператор `reduce` возвращает только один элемент, представляющий окончательный результат. Однако иногда при выполнении агрегирования желательно передавать дальше промежуточные результаты. Реализовать это можно с помощью оператора `Flux.scan()`. Например, подсчитаем сумму целых чисел от 1 до 10 с помощью оператора `scan`.

```
Flux.range(1, 5)
    .scan(0, (acc, elem) -> acc + elem)
    .subscribe(result -> log.info("Result: {}", result));
```

Этот код выведет следующие строки:

```
Result: 0
Result: 1
Result: 3
Result: 6
Result: 10
Result: 15
```

Как видите, окончательный результат получился тем же (15). Но в этот раз мы получили также промежуточные результаты. Оператор `scan` может найти применение в самых разных ситуациях, когда требуется иметь некоторую информацию о течении процесса. Например, вот как можно организовать вычисление скользящего среднего по потоку:

```
int bucketSize = 5; // (1)
Flux.range(1, 500) // (2)
    .index() // (3)
    .scan( // (4)
        new int[bucketSize], // (4.1)
        (acc, elem) -> { //
            acc[(int)(elem.getT1() % bucketSize)] = elem.getT2(); // (4.2)
            return acc; // (4.3)
        })
    .skip(bucketSize) // (5)
    .map(array -> Arrays.stream(array).sum() * 1.0 / bucketSize) // (6)
    .subscribe(av -> log.info("Running average: {}", av)); // (7)
```

Опишем работу этого кода.

1. Определяется размер скользящего окна для вычисления среднего значения (предполагается, что нас интересуют последние пять элементов).
2. С помощью оператора `range` генерируются некоторые данные.
3. С помощью оператора `index` к каждому элементу присоединяется его индекс.
4. С помощью оператора `scan` накапливаются последние пять элементов (4.1), при этом индексы элементов используются для определения позиции в контейнере (4.2). На каждом шаге возвращается тот же самый контейнер с обновленными данными.
5. Пропускаем несколько элементов с начала потока, чтобы накопить достаточный объем данных для вычисления скользящего среднего.
6. Чтобы получить скользящее среднее, делим сумму значений в контейнере на его размер.
7. Чтобы получить данные, нужно создать подписку.

Классы `Mono` и `Flux` имеют операторы `then`, `thenMany` и `thenEmpty`, которые завершаются с окончанием входящего потока. Они игнорируют входящие элементы и воспроизводят только сигналы завершения и ошибки. Эти операторы можно использовать для запуска новых потоков по завершении обработки предыдущих.

```
Flux.just(1, 2, 3)
    .thenMany(Flux.just(4, 5))
    .subscribe(e -> log.info("onNext: {}", e));
```

Лямбда-выражение в вызове метода `subscribe` получит только 4 и 5, даже притом что 1, 2 и 3 были сгенерированы и обработаны потоком.

Комбинирование реактивных потоков

Библиотека Project Reactor позволяет объединить множество входящих потоков в исходящий поток. Перечисленные ниже операторы имеют множество вариантов и выполняют следующие преобразования:

- оператор `concat` объединяет все источники, пересылая полученные элементы дальше. При объединении двух потоков этот оператор сначала извлекает и пересылает элементы из первого потока, а потом из второго;
- оператор `merge` сливает данные из входящих потоков в одну нисходящую последовательность. В отличие от оператора `concat`, данные из входящих потоков потребляются одновременно и могут чередоваться в нисходящем потоке;
- оператор `zip` подписывается на все входящие потоки, ждет, когда все они пришлют по одному элементу, и группирует их в один исходящий элемент. Подробно о работе оператора `zip` рассказывалось в главе 2 «Реактивное программирование в Spring. Основные понятия». Оператор `zip` из библиотеки Reactor может взаимодействовать не только с реактивными издателями, но и с контейнерами `Iterable`. Для этого можно использовать оператор `zipWithIterable`;
- оператор `combineLatest` действует подобно оператору `zip`, но генерирует каждое новое значение, как только получит хотя бы один элемент из входящих потоков.

Давайте объединим несколько потоков.

```
Flux.concat(
    Flux.range(1, 3),
    Flux.range(4, 2),
    Flux.range(6, 5)
).subscribe(e -> log.info("onNext: {}", e));
```

Как нетрудно догадаться, этот код сгенерирует значения от 1 до 10 ($[1, 2, 3] + [4, 5] + [6, 7, 8, 9, 10]$).

Пакетная обработка элементов потока

Библиотека Project Reactor поддерживает несколько методов пакетной обработки элементов (`Flux<T>`):

- метод **буферизации элементов** в контейнеры, такие как `List`. Исходящий поток имеет тип `Flux<List<T>>`;
- метод **кадрирования** элементов в поток потоков, такой как `Flux<Flux<T>>`. Имейте в виду, что подобный поток передает не значения элементов, а подпотоки, которые должны обрабатываться как самостоятельные потоки;
- метод **группировки** элементов по некоторому ключу в поток типа `Flux<GroupedFlux<K, T>>`. Каждый новый ключ генерирует новый экземп-

ляр `GroupedFlux`, и все элементы с данным ключом передаются через этот экземпляр класса `GroupFlux`.

Буферизацию и кадрирование можно организовать следующим образом:

- по количеству обрабатываемых элементов – например, для каждых 10 элементов;
- по интервалу времени – например, для каждого 5-минутного интервала;
- по некоторому условию – например, по каждому новому четному числу;
- по некоторому событию, поступившему из другого потока `Flux`.

Давайте реализуем буферизацию целых чисел по размеру списка 4.

```
Flux.range(1, 13)
    .buffer(4)
    .subscribe(e -> log.info("onNext: {}", e));
```

Предыдущий код выведет следующие строки:

```
onNext: [1, 2, 3, 4]
onNext: [5, 6, 7, 8]
onNext: [9, 10, 11, 12]
onNext: [13]
```

В выводе программы видим, что все элементы, кроме последнего, – это списки с размером 4. Последний элемент оказался списком с длиной 1, потому что это остаток от деления числа 13 на 4. Оператор `buffer` собирает несколько событий в коллекцию и передает ее как самостоятельный элемент следующему оператору в потоке. Оператор `buffer` удобно использовать для пакетной обработки, когда желательно сделать меньшее число запросов на получение коллекций элементов вместо большого числа запросов – по одному для каждого элемента. Например, вместо последовательной записи элементов в базу данных можно наполнять буфер в течение пары секунд и затем выполнить пакетную запись. Конечно, такое возможно только при соблюдении всех ограничений согласованности.

Для демонстрации оператора `window` реализуем разбиение последовательности чисел на кадры всякий раз, как будет встречаться простое число. Для этого используем вариант `windowUntil` оператора `window`. Конец текущего кадра он определяет с помощью заданного предиката, как показано ниже.

```
Flux<Flux<Integer>> windowedFlux = Flux.range(101, 20)           // (1)
    .windowUntil(this::isPrime, true);                          // (2)

windowedFlux.subscribe(window -> window                        // (3)
    .collectList()                                              // (4)
    .subscribe(e -> log.info("window: {}", e)));                // (5)
```

Пояснения к предыдущему коду.

1. Сначала генерируется последовательность из 20 целых чисел начиная со 101.
2. Указываем, что всякий раз, как встречается простое число, должен начаться новый кадр с элементами. Вторым аргументом оператора `windowUntil` определяется, где должна проводиться граница кадра – до или после элемента, удовлетворившего условиям предиката. В данном случае граница проводится так, что новое простое число оказывается в начале следующего кадра. Получающийся в результате поток имеет тип `Flux<Flux<Integer>>`.
3. Теперь можно подписаться на поток `windowedFlux`. Но дело в том, что каждый элемент потока `windowedFlux` сам является реактивным потоком. Поэтому для каждого кадра выполняем еще одно реактивное преобразование.
4. В нашем случае собираем элементы из каждого кадра в коллекцию с помощью оператора `collectList`, в результате чего каждый кадр преобразуется в тип `Mono<List<Integer>>`.
5. Для каждого элемента внутри `Mono` создаем отдельную подписку и регистрируем полученные события.

Предыдущий код выведет следующие строки:

```
window: []  
window: [101, 102]  
window: [103, 104, 105, 106]  
window: [107, 108]  
window: [109, 110, 111, 112]  
window: [113, 114, 115, 116, 117, 118, 119, 120]
```

Обратите внимание, что первое окно – пустое. Так получилось потому, что при создании исходного потока данных тут же было сгенерировано начальное окно. Затем в потоке появился первый элемент (число 101), являющийся простым числом, который вызвал создание нового кадра и соответственно закрытие текущего (с сигналом `onComplete`) без элементов в нем.

Конечно, эту задачу можно также решить с помощью оператора `buffer`. Оба оператора действуют очень похоже. Но `buffer` посылает коллекцию, только когда она будет закрыта, а оператор `window` передает события дальше по мере их поступления, что позволяет быстрее реагировать на изменения в потоке данных и конструировать более сложные процессы обработки.

Также есть возможность сгруппировать элементы реактивного потока по произвольному критерию с помощью оператора `groupBy`. Например, разделим последовательность целых чисел на четные и нечетные значения и будем следить за двумя последними значениями в каждой группе. Вот как это можно реализовать.

```
Flux.range(1, 7) // (1)  
  .groupBy(e -> e % 2 == 0 ? "Even" : "Odd") // (2)  
  .subscribe(groupFlux -> groupFlux // (3)
```

```
.scan(                                     // (4)
    new LinkedList<>(),                     // (4.1)
    (list, elem) -> {
        list.add(elem);                    // (4.2)
        if (list.size() > 2) {
            list.remove(0);                // (4.3)
        }
        return list;
    })
.filter(arr -> !arr.isEmpty())             // (5)
.subscribe(data ->                         // (6)
    log.info("{}: {}", groupFlux.key(), data));
```

Пояснения к коду.

1. Генерируется короткая последовательность чисел.
2. С помощью оператора `groupBy` последовательность разбивается на четные и нечетные числа, определение которых выполняется с использованием операции деления по модулю. Оператор возвращает поток типа `Flux<GroupedFlux<String, Integer>>`.
3. Выполняется подписка на главный поток `Flux`, и к каждому из сгруппированных потоков применяется оператор `scan`.
4. Оператор `scan` инициализируется пустым списком (4.1). Каждый элемент из сгруппированного потока добавляется в список (4.2), и если в списке оказалось больше двух элементов, самые старые элементы удаляются (4.3).
5. Оператор `scan` посылает начальное значение, а затем выполняет пересчет. В данном случае оператор `filter` позволяет удалить из потока пустые контейнеры, представляющие начальное значение `scan`.
6. В заключение выполняется подписка на каждый сгруппированный поток и вывод элементов, посылаемых оператором `scan`.

Предыдущий код выведет следующие строки:

```
Odd: [1]
Even: [2]
Odd: [1, 3]
Even: [2, 4]
Odd: [3, 5]
Even: [4, 6]
Odd: [5, 7]
```

Библиотека Project Reactor поддерживает также некоторые расширенные методы, такие как группировка элементов по разным кадрам во времени. Подробнее об этих возможностях можно узнать в документации с описанием оператора `groupJoin`.

Операторы flatMap, concatMap и flatMapSequential

Конечно, разработчики Project Reactor не могли пройти мимо реализации оператора flatMap, потому что он представляет собой важнейшее преобразование в функциональном программировании.

Оператор flatMap логически состоит из двух операций: *map* (отображение) и *flatten* (преобразование в плоское представление, в терминах Reactor эта операция напоминает оператор merge). Операция отображения (*map*) в операторе flatMap преобразует каждый входящий элемент в реактивный поток ($T \rightarrow \text{Flux}<R>$), а операция преобразования в плоское представление сливает сгенерированные реактивные последовательности в новую реактивную последовательность, передавая через нее элементы типа *R*.

Диаграмма на рис. 4.7 должна помочь понять суть этой идеи.

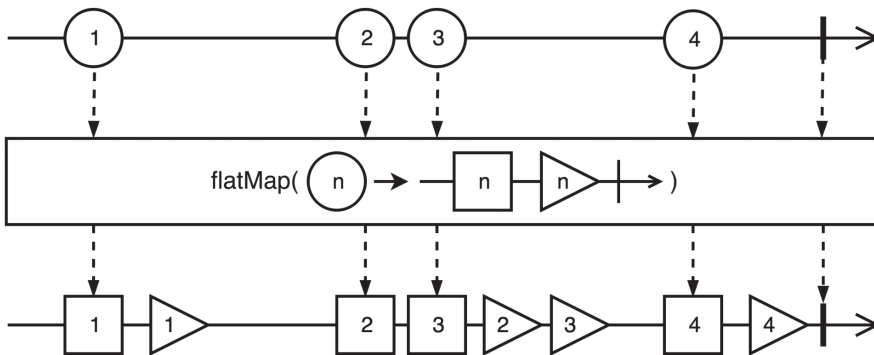


Рис. 4.7. Оператор flatMap

На диаграмме для каждого *кружка с цифрой* генерируется *квадратик с цифрой*, а затем *треугольник с цифрой*. Все такие подпоследовательности объединяются (сливаются) в общий исходящий поток.

Библиотека Project Reactor предлагает несколько дополнительных вариантов оператора flatMap, а также операторы flatMapSequential и concatMap. Эти три оператора имеют следующие отличия друг от друга:

- по-разному подписываются на внутренние потоки (flatMap и flatMap Sequential подписываются сразу на все потоки, а оператор concatMap ждет завершения очередного внутреннего потока, прежде чем сгенерировать следующий и подписаться на него);
- генерируют элементы в разном порядке (оператор concatMap естественным образом сохраняет исходный порядок элементов, оператор flatMap Sequential сохраняет исходный порядок элементов, полученных не по порядку, сохраняя их в очереди, а оператор flatMap не гарантирует сохранение исходного порядка элементов);

- по-разному перемежают элементы из разных потоков (оператор `flatMap` допускает смешивание элементов из разных потоков, тогда как операторы `concatMap` и `flatMapSequential` – нет).

Реализуем алгоритм, который запрашивает у каждого пользователя список его любимых книг. Служба, возвращающая список любимых книг пользователя, может выглядеть примерно так:

```
public Flux<String> requestBooks(String user) {  
    return Flux.range(1, random.nextInt(3) + 1)           // (1)  
        .map(i -> "book-" + i)                             // (2)  
        .delayElements(Duration.ofMillis(3));             // (3)  
}
```

Вот что делает эта вымышленная служба.

1. Генерирует набор случайных целочисленных значений.
2. Отображает каждое значение в название книги.
3. Приостанавливается после каждой книги, имитируя задержку, вызванную обращением к базе данных.

Теперь объединим результаты вызовов метода `requestBooks` для нескольких пользователей.

```
Flux.just("user-1", "user-2", "user-3")  
    .flatMap(u -> requestBooks(u)  
        .map(b -> u + "/" + b))  
    .subscribe(r -> log.info("onNext: {}", r));
```

Этот код сгенерирует следующие строки, наглядно показывающие, как происходит перемежение элементов:

```
[thread: parallel-3] onNext: user-3/book-1  
[thread: parallel-1] onNext: user-1/book-1  
[thread: parallel-1] onNext: user-2/book-1  
[thread: parallel-4] onNext: user-3/book-2  
[thread: parallel-5] onNext: user-2/book-2  
[thread: parallel-6] onNext: user-1/book-2  
[thread: parallel-7] onNext: user-3/book-3  
[thread: parallel-8] onNext: user-2/book-3
```

Кроме всего прочего здесь можно видеть, что оператор `flatMap` посылает элементы в порядке их поступления из разных потоков выполнения. Однако стандарт Reactive Streams гарантирует семантику *happens-before*. То есть даже когда элементы поступают из разных потоков выполнения, никакие два элемента не поступят одновременно. Этот аспект Project Reactor подробно рассматривается в разделе «Модель планирования потоков выполнения в Reactor».

Кроме того, библиотека позволяет задерживать сигналы `onError` с помощью операторов `flatMapDelayError`, `flatMapSequentialDelayError` и `concatMapDelayError`. Вдобавок к этому оператор `concatMapIterable` дает возможность выполнить аналогичную операцию, когда функция преобразования генерирует не реактивный поток, а итератор для каждого элемента. В этом случае перемежения элементов не происходит.

Оператор `flatMap` (и его варианты) играет очень важную роль в функциональном и реактивном программировании, позволяя реализовать сложные процессы обработки одной строкой кода.

Извлечение выборки элементов

В сценариях, когда требуется высокая пропускная способность, иногда имеет смысл обрабатывать только часть событий, применяя прием извлечения выборки. Для этого Reactor предлагает операторы `sample` и `sampleTimeout`. То есть из последовательности можно периодически извлекать отдельные элементы, соответствующие последнему значению, наблюдавшемуся во временном окне. Рассмотрим следующий код.

```
Flux.range(1, 100)
    .delayElements(Duration.ofMillis(1))
    .sample(Duration.ofMillis(20))
    .subscribe(e -> log.info("onNext: {}", e));
```

Он сгенерирует следующий вывод:

```
onNext: 13
onNext: 28
onNext: 43
onNext: 58
onNext: 73
onNext: 89
onNext: 100
```

Как видим, даже притом что каждую миллисекунду код последовательно генерирует элементы, подписчик получает только часть событий в соответствии с установленным пределом. Таким способом можно обеспечить пассивное ограничение частоты получения элементов, если для успешного решения задачи не нужно получить все события.

Преобразование реактивных последовательностей в блокирующие структуры

Библиотека Project Reactor предлагает API для преобразования реактивных последовательностей в блокирующие структуры. Даже притом что никакие блоки-

рующие операции нежелательны в реактивном приложении, иногда блокировка может требоваться программным интерфейсом верхнего уровня. По этой причине библиотека предлагает следующие варианты для блокировки и синхронизации потока данных:

- метод `toIterable` преобразует реактивный поток `Flux` в блокирующую структуру `Iterable`;
- метод `toStream` преобразует реактивный поток `Flux` в блокирующий `Stream` API. Начиная с версии `Reactor 3.2` он используется методом `toIterable`;
- метод `blockFirst` блокирует текущий поток выполнения, пока входящий поток данных не передаст первое значение или не завершится;
- метод `blockLast` блокирует текущий поток выполнения, пока входящий поток данных не передаст последнее значение или не завершится. В случае сигнала `onError` возбудит исключение в заблокированном потоке выполнения.

Важно помнить, что операторы `blockFirst` и `blockLast` имеют перегруженные версии, которые позволяют задать предельное время блокировки потока выполнения. С их помощью можно избежать блокировки потока выполнения «навечно». Кроме того, методы `toIterable` и `toStream` могут использовать очередь `Queue` для сохранения событий, если они поступают быстрее, чем клиент успевает выполнять блокирующие итерации с `Iterable` или `Stream`.

Просмотр элементов при обработке последовательности

Иногда для каждого элемента или конкретного сигнала требуется выполнить некоторое действие в середине конвейера обработки. Для таких случаев в `Project Reactor` имеются следующие методы:

- `doOnNext(Consumer<T>)` позволяет выполнить некое действие с каждым элементом в потоке данных `Flux` или `Mono`;
- `doOnComplete()` и `doOnError(Throwable)` вызываются в соответствующих обработчиках событий;
- `doOnSubscribe(Consumer<Subscription>)`, `doOnRequest(LongConsumer)` и `doOnCancel(Runnable)` позволяют определить реакцию на события жизненного цикла подписки;
- `doOnTerminate(Runnable)` вызывается с завершением потока данных независимо от причины, вызвавшей завершение.

Также `Flux` и `Mono` предлагают метод `doOnEach(Consumer<Signal>)`, который обрабатывает все сигналы реактивного потока – `onError`, `onSubscribe`, `onNext`, `onError` и `onComplete`.

Взгляните на следующий код.


```
Flux.just(1, 2, 3)
    .concatWith(Flux.error(new RuntimeException("Conn error")))
    .doOnEach(s -> log.info("signal: {}", s))
    .subscribe();
```

Здесь используется оператор `concatWith` – удобная обертка вокруг оператора `concat`. Этот код сгенерирует следующие строки:

```
signal: doOnEach_onNext(1)
signal: doOnEach_onNext(2)
signal: doOnEach_onNext(3)
signal: onError(java.lang.RuntimeException: Conn error)
```

В этом примере мы получаем не только все сигналы `onNext`, но и сигнал `onError`.

Материализация и дематериализация сигналов

Иногда потоки данных удобно обрабатывать не с позиции данных, а с позиции сигналов. Для преобразования потока данных в поток сигналов и обратно Flux и Mono предлагают методы `materialize` и `dematerialize`. Вот как можно их использовать.

```
Flux.range(1, 3)
    .doOnNext(e -> log.info("data : {}", e))
    .materialize()
    .doOnNext(e -> log.info("signal: {}", e))
    .dematerialize()
    .collectList()
    .subscribe(r -> log.info("result: {}", r));
```

Этот код выведет следующие строки:

```
data : 1
signal: onNext(1)
data : 2
signal: onNext(2)
data : 3
signal: onNext(3)
signal: onComplete()
result: [1, 2, 3]
```

Когда обрабатывается поток сигналов, метод `doOnNext` получает не только события `onNext` с данными, но и событие `onComplete`, звернутое в класс `Signal`. Такой подход позволяет обрабатывать события `onNext`, `onError` и `onComplete` в пределах одной иерархии типов.

Для случаев, когда нужно лишь регистрировать сигналы без изменения, Reactor предлагает метод `log`, который использует доступный механизм журналирования для регистрации всех обрабатываемых сигналов.

Поиск подходящего оператора

Project Reactor предлагает очень гибкий предметно-ориентированный язык (DSL) для обработки реактивных потоков. Однако, чтобы привыкнуть к библиотеке, требуется некоторая практика, поэтому выбор подходящего оператора не является проблемой. Гибкий API библиотеки Reactor и превосходная документация помогут вам в этом. Мы также советуем обращаться к разделу «*Which operator do I need?*» (*Какой оператор мне нужен?*) в официальной документации, когда возникают сомнения в выборе оператора для решения конкретной задачи. Текст раздела доступен по ссылке <http://projectreactor.io/docs/core/release/reference/#which-operator>.

Создание потоков данных программным способом

Мы уже знаем, как создавать реактивные потоки из массивов, объектов `Future` и блокирующих запросов. Но иногда требуется реализовать более сложный способ генерации сигналов в потоке или связать жизненный цикл объекта с жизненным циклом реактивного потока. В этом разделе рассмотрим инструменты, предлагаемые библиотекой Reactor для создания потоков данных программным способом.

Фабричные методы `push` и `create`

Фабричный метод `push` позволяет программно создать экземпляр Flux на основе однопоточного производителя. Этот подход можно использовать для адаптации асинхронного, однопоточного, многозначного API, не заботясь об управлении обратным давлением или отмене подписки. Оба аспекта управляются посредством сигналов, если подписчик не справляется с нагрузкой. Взгляните на следующий код.

```
Flux.push(emitter -> IntStream                // (1)
        .range(2000, 3000)                    // (1.1)
        .forEach(emitter::next))              // (1.2)
        .delayElements(Duration.ofMillis(1)) // (2)
        .subscribe(e -> log.info("onNext: {}", e)); // (3)
```

Пояснения к коду.

1. Вызывается фабричный метод `push`, адаптирующий некий существующий API под реактивную парадигму. Для простоты использован Java Stream API, с помощью которого генерируется 1000 целочисленных элементов (1.1) и передается объекту `emitter` типа `FluxSink` (1.2). Внутри метода мы никак не заботимся ни об обратном давлении, ни об отмене подписки, потому что эти функции реализует сам метод `push`.
2. Задержка после каждого элемента для имитации обратного давления.
3. Подписка на события `onNext`.

Метод `push` удобно использовать для адаптации асинхронного API со стратегиями обратного давления и отмены по умолчанию.

Также в нашем распоряжении имеется фабричный метод `create`, который действует подобно методу `push`, но дополнительно позволяет пересылать события из разных потоков выполнения, сериализуя экземпляр `FluxSink`. Оба метода допускают переопределение стратегии реакции на переполнение и освобождение ресурсов путем регистрации дополнительных обработчиков, как показано в следующем коде.

```
Flux.create(emitter -> {
    emitter.onDispose(() -> log.info("Disposed"));
    // отправить события объекту emitter
})
    .subscribe(e -> log.info("onNext: {}", e));
```

Фабричный метод `generate`

Фабричный метод `generate` задумывался с целью дать возможность генерировать сложные последовательности на основе внутреннего состояния генератора. Он требует передачи начального значения и функции, вычисляющей следующее внутреннее состояние на основе предыдущего, и посылает сигнал `onNext` подписчику на исходящий поток. Например, создадим простой реактивный поток, генерирующий последовательность чисел Фибоначчи (1, 1, 2, 3, 5, 8, 13, ...).

```
Flux.generate(                                     // (1)
    () -> Tuples.of(0L, 1L),                       // (1.1)
    (state, sink) -> {                             //
        log.info("generated value: {}", state.getT2()); //
        sink.next(state.getT2());                   // (1.2)
        long newValue = state.getT1() + state.getT2(); //
        return Tuples.of(state.getT2(), newValue);   // (1.3)
    })
    .delayElements(Duration.ofMillis(1))            // (2)
    .take(7)                                         // (3)
    .subscribe(e -> log.info("onNext: {}", e));      // (4)
```

Пояснения к коду.

1. С помощью фабричного метода `generate` можно создать свою реактивную последовательность. В качестве начального состояния используется `Tuples.of(0L, 1L)` (1.1). На этапе генерирования посылаем сигнал `onNext` со ссылкой на второе значение из пары, описывающей состояние (1.2), и вычисляем новую пару, опираясь на следующее значение в последовательности Фибоначчи (1.3).
2. С помощью оператора `delayElements` вводится некоторая задержка между сигналами `onNext`.
3. Ради простоты извлекаем только первые семь элементов.
4. Наконец, чтобы запустить генерирование последовательности, подписываемся на события.

Предыдущий код выведет следующие строки:

```
generated value: 1
onNext: 1
generated value: 1
onNext: 1
generated value: 2
onNext: 2
generated value: 3
onNext: 3
generated value: 5
onNext: 5
generated value: 8
onNext: 8
generated value: 13
onNext: 13
```

Как видите, каждое новое значение синхронно передается подписчику, и только после этого генерируется новое. Данный подход можно использовать для создания сложных реактивных последовательностей, требующих сохранения промежуточного состояния между созданием элементов.

Передача одноразовых ресурсов в реактивные потоки

С помощью фабричного метода `using` можно создать поток данных, использующий одноразовые ресурсы. Он реализует семантику `try-with-resources` в реактивном программировании. Предположим, потребовалось обернуть блокирующий API, представленный следующим преднамеренно упрощенным классом `Connection`.

```
public class Connection implements AutoCloseable {                                / (1)
```

```
private final Random rnd = new Random();

public Iterable<String> getData() { // (2)
    if (rnd.nextInt(10) < 3) { // (2.1)
        throw new RuntimeException("Communication error");
    }
    return Arrays.asList("Some", "data"); // (2.2)
}

public void close() { // (3)
    log.info("IO Connection closed");
}

public static Connection newConnection() { // (4)
    log.info("IO Connection created");
    return new Connection();
}
}
```

Пояснения к коду.

1. Класс `Connection` управляет некоторыми внутренними ресурсами и уведомляет об этом, реализуя интерфейс `AutoClosable`.
2. Метод `getData` имитирует операцию ввода/вывода и может вызывать исключения (2.1) или возвращать коллекцию `Iterable` с полезными данными (2.2).
3. Метод `close` может освобождать внутренние ресурсы и должен вызываться всегда, даже если в `getData` произойдет ошибка.
4. Статический фабричный метод `newConnection` всегда возвращает новый экземпляр класса `Connection`.

Обычно соединения и фабрики соединений имеют более сложное поведение, но для наглядности используем эту упрощенную реализацию.

При использовании императивного стиля программирования данные из соединения можно получить так, как показано ниже.

```
try (Connection conn = Connection.newConnection()) { // (1)
    conn.getData().forEach( // (2)
        data -> log.info("Received data: {}", data)
    );
} catch (Exception e) { // (3)
    log.info("Error: {}", e.getMessage());
}
```

Вот как действует этот код.

1. Использует Java-инструкцию `try` для создания нового соединения и автоматически закрывает его при выходе из текущего блока кода.
2. Получает и обрабатывает данные.
3. В случае исключения регистрирует сообщение об ошибке в журнале.

А так выглядит реактивный эквивалент предыдущего кода.

```
Flux<String> ioRequestResults = Flux.using(           // (1)
    Connection::newConnection,                       // (1.1)
    connection -> Flux.fromIterable(connection.getData()), // (1.2)
    Connection::close                               // (1.3)
);

ioRequestResults.subscribe(                          // (2)
    data -> log.info("Received data: {}", data),    //
    e -> log.info("Error: {}", e.getMessage()),     //
    () -> log.info("Stream finished"));              //
```

Пояснения к коду.

1. Фабричный метод `using` позволяет связать жизненный цикл экземпляра `Connection` с жизненным циклом обертывающего потока данных. Метод `using` должен знать, как создать ресурс. В данном случае это код создания нового соединения (1.1). Также метод должен знать, как преобразовать только что созданный ресурс в реактивный поток. В данном случае это вызов метода `fromIterable` (1.2). Наконец, он должен знать, как закрыть ресурс. В данном случае по завершении обработки должен быть вызван метод `close` экземпляра соединения.
2. Конечно, чтобы начать фактическую обработку, нужно создать подписку, обрабатывающую сигналы `onNext`, `onError` и `onComplete`.

В случае успешного выполнения предыдущий код выведет следующие строки:

```
IO Connection created
Received data: Some
Received data: data
IO Connection closed
Stream finished
```

Если симитировать ошибку, он выведет это:

```
IO Connection created
IO Connection closed
Error: Communication error
```

В обоих случаях оператор `using` сначала создает новое соединение, затем выполняет рабочую процедуру (успешно или нет) и закрывает соединение. В данном случае жизненный цикл соединения связан с жизненным циклом потока данных.

Также оператор позволяет выбирать, когда освободить ресурс – до уведомления подписчика о завершении потока данных или после.

Обертывание транзакций с помощью фабричного метода `usingWhen`

Похожий на `using` оператор `usingWhen` позволяет управлять ресурсами на реактивный манер. Однако если оператор `using` получает управляемый ресурс синхронно (вызовом экземпляра `Callable`), то оператор `usingWhen` получает управляемый ресурс реактивно (подписавшись на экземпляр `Publisher`). Кроме того, оператор `usingWhen` принимает разные обработчики для событий успешного или неудачного завершения обработки главного потока данных. Эти обработчики реализуются издателями. Данное отличие помогает реализовать неблокирующие реактивные транзакции одним оператором.

Допустим, у нас есть полностью реактивная транзакция. Код реализации сильно упрощен.

```
public class Transaction {
    private static final Random random = new Random();
    private final int id;

    public Transaction(int id) {
        this.id = id;
        log.info("[T: {}] created", id);
    }

    public static Mono<Transaction> beginTransaction() {           // (1)
        return Mono.defer(() ->
            Mono.just(new Transaction(random.nextInt(1000))));
    }

    public Flux<String> insertRows(Publisher<String> rows) {      // (2)
        return Flux.from(rows)
            .delayElements(Duration.ofMillis(100))
            .flatMap(r -> {
                if (random.nextInt(10) < 2) {
                    return Mono.error(new RuntimeException("Error: " + r));
                } else {
                    return Mono.just(r);
                }
            });
    }

    public Mono<Void> commit() {                                   // (3)
        return Mono.defer(() -> {
```

```

        log.info("[T: {}] commit", id);
        if (random.nextBoolean()) {
            return Mono.empty();
        } else {
            return Mono.error(new RuntimeException("Conflict"));
        }
    });
}

public Mono<Void> rollback() { // (4))
    return Mono.defer(() -> {
        log.info("[T: {}] rollback", id);
        if (random.nextBoolean()) {
            return Mono.empty();
        } else {
            return Mono.error(new RuntimeException("Conn error"));
        }
    });
}
}

```

Пояснения к коду.

1. Это статический фабричный метод создания новой транзакции.
2. Каждая транзакция имеет метод сохранения новых записей в пределах транзакции. Иногда процесс может терпеть неудачу из-за каких-то внутренних проблем (здесь реализуется как случайное поведение). `insertRows` потребляет и возвращает реактивные потоки.
3. Это асинхронный метод `commit` подтверждения транзакции. Иногда попытка подтверждения терпит неудачу.
4. Асинхронный метод `rollback` отката транзакции. Иногда попытка отката может потерпеть неудачу.

Теперь воспользуемся оператором `usingWhen` и реализуем обновление данных с использованием транзакции:

```

Flux.usingWhen(
    Transaction.beginTransaction(), // (1)
    transaction -> transaction.insertRows(Flux.just("A", "B", "C")), // (2)
    Transaction::commit, // (3)
    Transaction::rollback // (4)
).subscribe(
    d -> log.info("onNext: {}", d),
    e -> log.info("onError: {}", e.getMessage()),
    () -> log.info("onComplete")
);

```


Предыдущий код использует оператор `usingWhen` для выполнения следующих операций.

1. Статический метод `beginTransaction` асинхронно возвращает новую транзакцию посредством типа `Mono<Transaction>`.
2. С помощью данного экземпляра транзакции предпринимается попытка вставить новые записи.
3. Если шаг (2) выполнен успешно, производится подтверждение транзакции.
4. Если шаг (2) потерпел неудачу, производится откат транзакции.

Выполнив этот код, вы увидите такой вывод (в случае успешного выполнения):

```
[T: 265] created
onNext: A
onNext: B
onNext: C
[T: 265] commit
onComplete
```

В случае неудачи вывод будет выглядеть немного иначе:

```
[T: 582] created
onNext: A
[T: 582] rollback
onError: Error: B
```

Оператор `usingWhen` значительно упрощает управление жизненным циклом ресурса в реактивной парадигме. С его помощью легко реализовать реактивные транзакции. Поэтому оператор `usingWhen` можно рассматривать как более совершенную версию оператора `using`.

Обработка ошибок

Проектируя реактивное приложение, которое интенсивно взаимодействует с внешними службами, мы имеем дело со всеми видами исключительных ситуаций. К счастью, сигнал `onError` является неотъемлемой частью стандарта Reactive Streams, поэтому всегда должен быть способ передачи исключения актору, который обработает его. Однако если конечный подписчик не определит обработчика для сигнала `onError`, то в ответ на него будет возбуждено исключение `UnsupportedOperationException`.

Кроме того, семантика реактивных потоков требует, чтобы сигнал `onError` завершал поток, то есть после него обработка последовательности должна прекращаться. В этот момент мы можем по-разному среагировать на сигнал, применив одну из следующих стратегий:

- безусловно, мы должны определить обработчик сигнала onError в операторе subscribe;
- можно перехватить и подменить ошибку статическим значением по умолчанию или значением, вычисленным на основе исключения, применив оператор onErrorReturn;
- можно перехватить исключение и выполнить альтернативную процедуру с помощью оператора onErrorResume;
- можно перехватить исключение и преобразовать его в другое исключение, лучше описывающее ситуацию, применив оператор onErrorMap;
- можно определить реактивный процесс так, чтобы в случае ошибки он повторил попытку выполнения. Оператор retry создаст повторную подписку на источник реактивной последовательности, если тот сообщит об ошибке. Он может повторять попытки до бесконечности или ограничиваться некоторым интервалом времени. Оператор retryBackoff предлагает готовую поддержку алгоритма экспоненциальной задержки, который постепенно увеличивает задержки между повторными попытками.

Кроме того, нередко пустой поток данных совсем не то, что требуется. В таком случае можно вернуть значение по умолчанию, воспользовавшись оператором defaultIfEmpty или запустив другой реактивный поток данных оператором switchIfEmpty.

Еще один удобный оператор – timeout, позволяет ограничить период ожидания и возбудить исключение TimeoutException, которое, в свою очередь, можно обрабатывать, применив ту или иную стратегию обработки ошибок.

Давайте посмотрим, как можно использовать некоторые из представленных стратегий. Допустим, у нас имеется следующая ненадежная служба подбора рекомендаций.

```
public Flux<String> recommendedBooks(String userId) {  
    return Flux.defer(() -> {                                // (1)  
        if (random.nextInt(10) < 7) {  
            return Flux.<String>error(new RuntimeException("Err")) // (2)  
                .delaySequence(Duration.ofMillis(100));  
        } else {  
            return Flux.just("Blue Mars", "The Expanse") // (3)  
                .delayElements(Duration.ofMillis(50));  
        }  
    }).doOnSubscribe(s -> log.info("Request for {}", userId)); // (4)  
}
```

Вот что делает этот код.

1. Откладывает вычисления, пока не появится подписчик.

2. Весьма вероятно, что наша служба вернет ошибку, поэтому мы сдвигаем все сигналы во времени, применив оператор `delaySequence`.
3. Если клиенту повезет, он получит свои рекомендации с некоторой задержкой.
4. Ответ службы записывается в журнал.

Теперь реализуем клиента этой ненадежной службы.

```
Flux.just("user-1") // (1)
  .flatMap(user -> // (2)
    recommendedBooks(user) // (2.1)
    .retryBackoff(5, Duration.ofMillis(100)) // (2.2)
    .timeout(Duration.ofSeconds(3)) // (2.3)
    .onErrorResume(e -> Flux.just("The Martian"))) // (2.4)
  .subscribe( // (3)
    b -> log.info("onNext: {}", b),
    e -> log.warn("onError: {}", e.getMessage()),
    () -> log.info("onComplete")
  );
```

Пояснения к коду.

1. Генерируется поток пользователей, попросивших рекомендовать фильмы для просмотра.
2. Для каждого пользователя вызывается наша ненадежная служба `recommendedBooks` (2.1). Если вызов терпит неудачу, мы повторяем попытки, постепенно увеличивая задержку (выполняется не более пяти попыток, задержки начинаются со 100 мс) (2.2). Если стратегия повторных попыток не дала результатов спустя три секунды, генерируется сигнал ошибки (2.3). Наконец, в случае любой ошибки с помощью оператора `onErrorResume` возвращаются предопределенные универсальные рекомендации (2.4).
3. И конечно, мы должны создать подписчика.

Запустив это приложение, мы получили следующие результаты:

```
[time: 18:49:29.543] Request for user-1
[time: 18:49:29.693] Request for user-1
[time: 18:49:29.881] Request for user-1
[time: 18:49:30.173] Request for user-1
[time: 18:49:30.972] Request for user-1
[time: 18:49:32.529] onNext: The Martian
[time: 18:49:32.529] onComplete
```

Как видим, наш клиент пять раз пытался получить рекомендации для `user-1`. В ходе попыток задержка увеличилась с ~150 мс до ~1,5 секунды. Наконец, наш код прекратил попытки получить результаты от метода `recommendedBooks`, вернул рекомендацию по умолчанию (`The Martian`) и завершил поток данных.

Подводя итог, отметим, что Project Reactor предлагает широкий спектр инструментов, помогающих обрабатывать исключительные ситуации и соответственно повысить устойчивость приложения.

Управление обратным давлением

Несмотря на требование стандарта Reactive Stream встраивать управление обратным давлением в механизм взаимодействий между производителем и потребителем, иногда возможны ситуации переполнения потребителя. Некоторые потребители могут нечаянно запросить неограниченный объем данных и затем не справиться с полученной нагрузкой. Отдельные потребители могут иметь жесткие ограничения на скорость поступления сообщений. Например, клиенту базы данных может быть не позволено сделать более 1000 записей в секунду. В таких случаях пригодятся приемы пакетирования. Мы рассмотрим этот подход в разделе «Пакетирование элементов потока». С другой стороны, можно настроить обработку обратного давления в потоке данных, используя следующие инструменты:

- оператор `onBackPressureBuffer` запрашивает неограниченный объем данных и передает получаемые элементы в нисходящий поток, а если потребитель нисходящего потока не справляется с нагрузкой, он сохраняет элементы в очереди. Оператор `onBackPressureBuffer` имеет множество перегруженных версий и предлагает большое число параметров настройки для управления поведением;
- оператор `onBackPressureDrop` также запрашивает неограниченный объем данных (`Integer.MAX_VALUE`) и передает получаемые элементы в нисходящий поток. Если потребитель нисходящего потока не справляется с нагрузкой, оператор отбрасывает элементы. Такие отбрасываемые элементы можно обрабатывать с помощью дополнительного обработчика;
- оператор `onBackPressureLast` действует подобно `onBackPressureDrop`, но сохраняет последний отброшенный элемент и передает его в нисходящий поток, как только появляется возможность. Таким способом всегда можно получить самый последний элемент – даже в ситуации переполнения;
- оператор `onBackPressureError` запрашивает неограниченный объем данных и пытается передать их в нисходящий поток. Если потребитель нисходящего потока не справляется с нагрузкой, оператор возбуждает ошибку.

Другой прием управления обратным давлением – ограничение скорости. Оператор `limitRate(n)` разбивает запрошенный снизу объем данных на пакеты с размером не больше `n`. С помощью этого оператора можно защитить уязвимо-го производителя от запроса необоснованно большого объема данных. Оператор `limitRequest(n)` позволяет ограничить требование (общее количество запрашиваемых элементов) со стороны потребителя. Например, вызов `limitRequest(100)` гарантирует, что производитель не получит запроса, требующего послать больше

100 элементов. После отправки 100 элементов оператор благополучно закроет поток данных.

Горячие и холодные потоки данных

Реактивные издатели делятся на два типа: **горячие** и **холодные**.

При появлении подписчика холодный издатель генерирует новую последовательность. Кроме того, он не генерирует никаких данных, если нет подписчика. Например, следующий код иллюстрирует поведение холодного издателя.

```
Flux<String> coldPublisher = Flux.defer(() -> {
    log.info("Generating new items");
    return Flux.just(UUID.randomUUID().toString());
});

log.info("No data was generated so far");
coldPublisher.subscribe(e -> log.info("onNext: {}", e));
coldPublisher.subscribe(e -> log.info("onNext: {}", e));
log.info("Data was generated twice for two subscribers");
```

Этот код выведет следующие строки:

```
No data was generated so far
Generating new items
onNext: 63c8d67e-86e2-48fc-80a8-a9c039b3909c
Generating new items
onNext: 52232746-9b19-4b5e-b6b9-b0a2fa76079a
Data was generated twice for two subscribers
```

Как видите, когда появляется подписчик, для него генерируется новая последовательность. Это напоминает семантику HTTP-запросов. Вызов не происходит, пока кто-то не заинтересуется результатом, и для каждого нового подписчика выполняется свой HTTP-запрос.

Горячие издатели, напротив, генерируют данные независимо от наличия подписчиков. То есть горячий издатель может начать генерировать элементы до появления первого подписчика. Кроме того, когда появляется первый подписчик, горячий издатель может не посылать ему ранее сгенерированные данные – посылает только новые. Такую семантику можно сравнить со сценарием широковещательной рассылки данных. Например, горячий издатель может рассылать своим подписчикам текущие цены на нефть по мере их изменения. Однако вновь появившийся подписчик получит только новые данные, он не получит старых цен. Большинство горячих издателей в библиотеке Reactor реализует интерфейс `Processor`. Подробнее о процессорах Reactor рассказывается в разделе «*Процессы*».

Фабричный метод `just` создает горячего издателя, так как его значения вычисляются только один раз, когда создается издатель, и не пересчитываются при появлении новых подписчиков.

Экземпляр, возвращаемый методом `just`, можно преобразовать в холодного издателя, завернув его в вызов `defer`. В этом случае, даже притом что `just` генерирует значения в момент инициализации, сама инициализация случится, только когда появится новый подписчик. Последнее поведение диктуется фабричным методом `defer`.

Широковещательная рассылка элементов потока данных

Конечно, есть возможность превратить холодного издателя в горячего, применив реактивное преобразование. Например, нам может понадобиться рассылать результаты работы холодного процессора по мере их готовности нескольким подписчикам и избежать создания данных для каждого подписчика в отдельности. Для этого в Project Reactor имеется `ConnectableFlux`. Класс `ConnectableFlux` способен немедленно рассылать данные быстрым подписчикам и кешировать их для остальных, чтобы все могли обрабатывать эти данные в удобном для себя темпе. Размер очереди и тайм-ауты настраиваются с помощью методов `publish` и `replay` этого класса. Также `ConnectableFlux` может автоматически отслеживать число подписчиков и инициировать выполнение по достижении установленного порога при вызове методов `connect`, `autoConnect(n)`, `refCount(n)` и `refCount(int, Duration)`.

Рассмотрим пример, иллюстрирующий поведение этого класса.

```
Flux<Integer> source = Flux.range(0, 3)
    .doOnSubscribe(s ->
        log.info("new subscription for the cold publisher"));

ConnectableFlux<Integer> conn = source.publish();

conn.subscribe(e -> log.info("[Subscriber 1] onNext: {}", e));
conn.subscribe(e -> log.info("[Subscriber 2] onNext: {}", e));

log.info("all subscribers are ready, connecting");
conn.connect();
```

Если запустить этот код, он выведет следующие строки:

```
all subscribers are ready, connecting
new subscription for the cold publisher
[Subscriber 1] onNext: 0
[Subscriber 2] onNext: 0
```

```
[Subscriber 1] onNext: 1
[Subscriber 2] onNext: 1
[Subscriber 1] onNext: 2
[Subscriber 2] onNext: 2
```

Как видите, наш холодный издатель принял подписку и сгенерировал элементы только один раз, однако оба подписчика получили полный набор событий.

Кеширование элементов потока

С помощью `ConnectableFlux` легко реализовать разные стратегии кеширования данных. Но в Reactor уже есть API для кеширования событий – оператор `cache`. Внутренне этот оператор использует `ConnectableFlux`, следовательно, главная ценность такого решения – простой и удобный API. Объем данных, хранимых в кеше, можно настраивать, а также можно определить предельное время хранения каждого элемента в кеше. Посмотрим на примере, как работает этот механизм.

```
Flux<Integer> source = Flux.range(0, 2) // (1)
    .doOnSubscribe(s ->
        log.info("new subscription for the cold publisher"));

Flux<Integer> cachedSource = source.cache(Duration.ofSeconds(1)); // (2)

cachedSource.subscribe(e -> log.info("[S 1] onNext: {}", e)); // (3)
cachedSource.subscribe(e -> log.info("[S 2] onNext: {}", e)); // (4)

Thread.sleep(1200); // (5)
cachedSource.subscribe(e -> log.info("[S 3] onNext: {}", e)); // (6)
```

Вот что делает этот код.

1. Сначала создается холодный издатель, генерирующий несколько элементов.
2. С помощью оператора `cache` производится кеширование издателя на 1 секунду.
3. Подключается первый подписчик.
4. Сразу за ним подключается второй подписчик.
5. Приостановка на время, чтобы срок хранения данных в кеше истек.
6. Наконец, подключается третий подписчик.

Теперь посмотрите на вывод программы.

```
new subscription for the cold publisher
[S 1] onNext: 0
[S 1] onNext: 1
[S 2] onNext: 0
[S 2] onNext: 1
```

```
new subscription for the cold publisher
```

```
[S 3] onNext: 0
```

```
[S 3] onNext: 1
```

Судя по этому выводу, первые два подписчика получили одни данные, кешированные первой подпиской. Затем, после задержки, третий подписчик не смог получить кешированных данных, поэтому была создана новая подписка для холодного издателя. В конце третий подписчик получил запрошенные данные, хотя и не из кеша.

Совместное использование элементов из потока

Выше мы реализовали рассылку событий нескольким подписчикам, используя класс `ConnectableFlux`. В том примере мы сначала дождались появления подписчиков и только потом приступили к обработке. Оператор `share` преобразует холодного издателя в горячего и не передает каждому новому подписчику события, которые тот пропустил. Взгляните на следующий код.

```
Flux<Integer> source = Flux.range(0, 5)
    .delayElements(Duration.ofMillis(100))
    .doOnSubscribe(s ->
        log.info("new subscription for the cold publisher"));
```

```
Flux<Integer> cachedSource = source.share();
```

```
cachedSource.subscribe(e -> log.info("[S 1] onNext: {}", e));
```

```
Thread.sleep(400);
```

```
cachedSource.subscribe(e -> log.info("[S 2] onNext: {}", e));
```

Предыдущий код объявляет разделяемым холодный поток, генерирующий события каждые 100 мс, затем создает пару подписчиков и подписывает их на разделяемый поток с некоторым интервалом. Давайте посмотрим, как выглядит вывод приложения.

```
new subscription for the cold publisher
```

```
[S 1] onNext: 0
```

```
[S 1] onNext: 1
```

```
[S 1] onNext: 2
```

```
[S 1] onNext: 3
```

```
[S 2] onNext: 3
```

```
[S 1] onNext: 4
```

```
[S 2] onNext: 4
```

Как видно из результатов, первый подписчик начал принимать события с самого начала, а второй пропустил события, сгенерированные до его появления (и получил только события 3 и 4).

Работа со временем

Реактивное программирование предусматривает асинхронное выполнение операций, поэтому оно изначально предполагает наличие оси времени.

С помощью Project Reactor можно генерировать события, исходя из продолжительности, используя оператор `interval`, задерживать элементы, используя оператор `delayElements`, и задерживать все сигналы, используя оператор `delaySequence`. В данной главе мы уже использовали некоторые из этих операторов.

Мы уже видели, как можно осуществлять буферизацию и кадрирование по времени (операторы `buffer(Duration)` и `window(Window)`). Однако библиотека Reactor позволяет также реагировать на некоторые события, связанные со временем, например с помощью ранее описанных операторов `timestamp` и `timeout`. Оператор `elapsed`, как и `timestamp`, отмеряет интервал времени от предыдущего события. Рассмотрим следующий код.

```
Flux.range(0, 5)
    .delayElements(Duration.ofMillis(100))
    .elapsed()
    .subscribe(e -> log.info("Elapsed {} ms: {}", e.getT1(), e.getT2()));
```

Он генерирует события каждые 100 мс, как показано ниже:

```
Elapsed 151 ms: 0
Elapsed 105 ms: 1
Elapsed 105 ms: 2
Elapsed 103 ms: 3
Elapsed 102 ms: 4
```

Как видим, интервал между событиями выдерживается не совсем точно. Это обусловлено тем, что для планирования событий Reactor использует Java-класс `ScheduledExecutorService`, который сам по себе не гарантирует высокой точности. Поэтому будьте внимательны и не ждите от библиотеки Reactor значительной точности измерения интервалов времени.

Компоновка и преобразование реактивных потоков

При создании сложных реактивных процессов часто бывает нужно использовать одну и ту же последовательность операторов в нескольких местах. С помощью оператора `transform` можно перенести такие общие последовательности в отдельные объекты и повторно использовать их по мере необходимости. До сих пор

мы преобразовывали события внутри потока. С помощью оператора `transform` можно расширить саму структуру потока. Взгляните на следующий пример.

```
Function<Flux<String>, Flux<String>> logUserInfo =           // (1)
    stream -> stream                                         //
    .index()                                                 // (1.1)
    .doOnNext(tp ->                                         // (1.2)
        log.info("[{}] User: {}", tp.getT1(), tp.getT2())) //
    .map(Tuple2::getT2);                                     // (1.3)

Flux.range(1000, 3)                                         // (2)
    .map(i -> "user-" + i)                                   //
    .transform(logUserInfo)                                  // (3)
    .subscribe(e -> log.info("onNext: {}", e));
```

Пояснения к предыдущему коду.

1. Определение функции `logUserInfo` с сигнатурой `Function<Flux<String>, Flux<String>>`. Она преобразует реактивный поток строк в другой реактивный поток, который тоже генерирует строковые значения. В этом примере для каждого сигнала `onNext` наша функция записывает в журнал информацию о пользователе (1.2) и дополнительно нумерует входящие события с помощью оператора `index` (1.1). Исходящий поток не содержит никакой информации о нумерации, потому что номера удаляются с помощью вызова `map(Tuple2::getT2)` (1.3).
2. Генерируется несколько числовых идентификаторов пользователей.
3. Преобразование, определяемое функцией `logUserInfo`, встраивается в поток с помощью оператора `transform`.

Выполнив этот код, вы получите следующие результаты:

```
[0] User: user-1000
onNext: user-1000
[1] User: user-1001
onNext: user-1001
[2] User: user-1002
onNext: user-1002
```

Как видим, каждый элемент регистрируется функцией `logUserInfo` и конечным подписчиком, но функция `logUserInfo` дополнительно записывает порядковые номера событий.

Оператор `transform` меняет поведение потока только один раз – на этапе сборки потока. В то же время в библиотеке Reactor имеется оператор `compose`, который выполняет то же самое преобразование потока при появлении каждого нового подписчика. Давайте посмотрим, как он работает, на примере следующего кода.

```
Function<Flux<String>, Flux<String>> logUserInfo = (stream) -> { // (1)
    if (random.nextBoolean()) {
        return stream
            .doOnNext(e -> log.info("[path A] User: {}", e));
    } else {
        return stream
            .doOnNext(e -> log.info("[path B] User: {}", e));
    }
};

Flux<String> publisher = Flux.just("1", "2") // (2)
    .compose(logUserInfo); // (3)

publisher.subscribe(); // (4)
publisher.subscribe();
```

Пояснения к коду.

1. Так же как в предыдущем примере, здесь определяется функция преобразования. В данном случае функция случайно выбирает путь преобразования потока. Два возможных пути отличаются только префиксом в выводимом сообщении.
2. Создается издатель, который генерирует некоторые данные.
3. С помощью оператора `compose` функция `logUserInfo` встраивается в процесс обработки.
4. Здесь мы создаем двух подписчиков в надежде увидеть разное поведение для разных подписок.

Если запустить предыдущий код, он выведет следующие строки:

```
[path B] User: 1
[path B] User: 2
[path A] User: 1
[path A] User: 2
```

Сообщения подтверждают, что первая подписка инициировала путь `path B`, а вторая – `path A`. Конечно, оператор `compose` позволяет реализовать гораздо более сложную бизнес-логику, чем случайный выбор префикса сообщений. Оба оператора, `transform` и `compose`, дают мощную возможность повторного использования кода в реактивных приложениях.

Процессоры

Стандарт Reactive Streams определяет интерфейс `Processor`, который одновременно является и издателем `Publisher`, и подписчиком `Subscriber`. То есть мож-

но подписаться на события, генерируемые экземпляром `Processor`, и посылать ему свои сигналы (`onNext`, `onError`, `onComplete`). Авторы библиотеки Reactor рекомендуют стараться не пользоваться процессорами, потому что они сложны в применении и чреваты ошибками. В большинстве случаев процессоры можно заменить комбинацией операторов. Например, для адаптации сторонних API лучше использовать фабричные методы генераторов (`push`, `create`, `generate`).

Reactor предлагает следующие виды процессоров:

- **непосредственные** процессоры могут посылать данные только вручную, работая с получателем. Яркими представителями этой группы процессоров служат `DirectProcessor` и `UnicastProcessor`. `DirectProcessor` не поддерживает управление обратным давлением, но может использоваться для отправки событий нескольким подписчикам. `UnicastProcessor` поддерживает обратное давление, используя для этого внутреннюю очередь, но может обслужить не более одного подписчика;
- **синхронные** процессоры (`EmitterProcessor` и `ReplayProcessor`) могут посылать данные и вручную, и по подписке на вышестоящего издателя `Publisher`. `EmitterProcessor` способен обслуживать несколько подписчиков и выполнять их требования, но получать данные способен только от одного издателя `Publisher`, причем в синхронном режиме. `ReplayProcessor` действует подобно `EmitterProcessor`, но поддерживает несколько стратегий кеширования входящих данных;
- **асинхронные** процессоры (`WorkQueueProcessor` и `TopicProcessor`) могут пересылать данные вниз, получая их от нескольких вышестоящих издателей. Для этого они используют структуру данных `RingBuffer`. Асинхронные процессоры предлагают выделенный API для их создания, потому что число параметров настройки настолько велико, что их сложно инициализировать вручную, ничего не забыв. `TopicProcessor` совместим со стандартом `Reactive Streams` и для обслуживания каждого нижестоящего подписчика запускает отдельный поток выполнения. Как следствие число подписчиков, которых он сможет обслужить, ограничено. `WorkQueueProcessor` имеет похожие характеристики, однако он ослабляет некоторые требования `Reactive Streams`, что позволяет ему уменьшить потребление ресурсов.

Тестирование и отладка Project Reactor

Вместе с библиотекой Reactor распространяется гибкий и универсальный фреймворк тестирования. Библиотека `io.projectreactor:reactor-test` предлагает все инструменты, необходимые для тестирования реактивных процессов, реализованных с использованием Project Reactor. Подробнее о методах тестирования в реактивном программировании рассказывается в главе 9 «Тестирование реактивных приложений».

Реактивный код сложен в отладке, поэтому Project Reactor предусматривает методы, упрощающие процесс отладки. Так же как в любой платформе, основанной на обратных вызовах, трассировка стека в Project Reactor почти не дает полезной информации. Трассировка не позволяет определить точное место в коде, где возникло исключение. Библиотека Reactor включает механизм отладки для этапа сборки. Подробнее об этапе сборки в жизненном цикле потоков данных рассказывается в разделе «*Продвинутые средства в Project Reactor*». Активировать этот механизм можно с помощью следующего кода.

```
Hooks.onOperatorDebug();
```

После включения этот механизм начинает собирать трассировки стека для всех потоков данных, и позднее данная информация помогает ускорить поиск проблем. Однако процедура трассировки обходится очень дорого, поэтому пользоваться этим механизмом следует только в самом крайнем случае. За более подробной информацией обращайтесь к документации для библиотеки Reactor.

Кроме того, типы Flux и Mono из Project Reactor имеют удобный метод log. Он регистрирует сигналы, проходящие через оператор. Большое количество настроек этого метода дает достаточно свободы для получения необходимых сведений даже в ситуации отладки.

Дополнения к Reactor

Project Reactor – гибкая и богатая возможностями библиотека. Но никакая библиотека не может вместить в себя все утилиты, какие только могут понадобиться, поэтому появилось несколько проектов, расширяющих возможности Reactor в разных направлениях. Официальный проект Reactor Addons (<https://github.com/reactor/reactor-addons>) включает несколько модулей для проекта Reactor. На момент написания этих строк проект Reactor Addons включал следующие модули: reactor-adapter, reactor-logback и reactor-extra.

Модуль reactor-adapter реализует адаптеры для реактивных типов RxJava 2 и планировщики. Также этот модуль включает поддержку интеграции с Akka.

Модуль reactor-logback содержит средства высокоскоростного, асинхронного журналирования. Он основан на AsyncAppender из Logback и RingBuffer из LMAX Disruptor, интегрированных в класс Processor.

Модуль reactor-extra содержит дополнительные утилиты для удовлетворения продвинутых потребностей. Например, в модуле имеется класс TupleUtils, упрощающий использование класса Tuple. Мы расскажем, как пользоваться этим классом, в главе 7 «*Реактивный доступ к базам данных*». Также в модуле имеется класс MathFlux, который может вычислять сумму, среднее, максимальное

и минимальные значения в числовых потоках. Класс `ForkJoinPoolScheduler` служит адаптером для Java-класса `ForkJoinPool`, преобразуя его в Reactor-тип `Scheduler`. Добавить модуль в свой проект Gradle можно с помощью следующей инструкции импортирования.

```
compile 'io.projectreactor.addons:reactor-extra:3.2.RELEASE'
```

Кроме того, в экосистеме Project Reactor имеются реактивные драйверы для популярных асинхронных фреймворков и брокеров сообщений.

Модуль `The Reactor RabbitMQ` (<https://github.com/reactor/reactor-rabbitmq>) предлагает реактивного Java-клиента для RabbitMQ с удобным интерфейсом Reactor API. Модуль обеспечивает возможность асинхронного и неблокирующего обмена сообщениями с поддержкой обратного давления. Также он позволяет приложениям использовать RabbitMQ в роли шины сообщений совместно с типами `Flux` и `Mono`. Модуль `Reactor Kafka` (<https://github.com/reactor/reactor-kafka>) предлагает похожие возможности для брокера сообщений `Kafka`.

Одно из наиболее популярных расширений библиотеки Reactor называется `Reactor Netty` (<https://github.com/reactor/reactor-netty>). Оно позволяет преобразовать клиента и сервер `Netty TCP/HTTP/UDP` в реактивные типы Reactor. Модуль `Spring WebFlux` использует клиента `Reactor Netty` для создания неблокирующих веб-приложений. Более подробно об этом рассказывается в главе 6 «Неблокирующие и асинхронные взаимодействия с `WebFlux`».

Продвинутые средства в Project Reactor

В предыдущем разделе мы исследовали реактивные типы и операторы, используемые для реализации самых разных реактивных процессов. Теперь займемся более глубоким исследованием жизненного цикла реактивных потоков данных, особенностей многопоточного выполнения и внутренних оптимизаций в Project Reactor.

Жизненный цикл реактивных потоков данных

Чтобы понять, как осуществляется многопоточное выполнение и какие внутренние оптимизации реализованы в Reactor, нужно разобраться с жизненным циклом реактивных типов в Reactor.

Этап сборки

Жизненный цикл потока данных начинается с **этапа сборки**. Как сказано в предыдущих разделах, Reactor предлагает гибкий API, позволяющий конструировать

сложные конвейеры обработки элементов. На первый взгляд кажется, что API, предлагаемый библиотекой Reactor, напоминает построителя, конструирующего поток обработки из выбранных операторов. Как вы наверняка помните, шаблон проектирования «Построитель» (Builder) предполагает изменение промежуточных объектов в памяти в процессе подготовки и окончательную сборку нового объекта завершающей операцией, такой как `build`. В отличие от этого распространенного шаблона, Reactor API предлагает подход без изменения имеющихся объектов. Как следствие каждый применяемый оператор создает новый объект. В реактивных библиотеках этот процесс конструирования конвейера обработки называется **сборкой** (assembling). Чтобы было понятнее, следующий псевдокод демонстрирует, как могла бы выглядеть сборка потока обработки без Reactor API.

```
Flux<Integer> sourceFlux = new FluxArray(1, 20, 300, 4000);
Flux<String> mapFlux = new FluxMap(sourceFlux, String::valueOf);
Flux<String> filterFlux = new FluxFilter(mapFlux, s -> s.length() > 1)
...
```

Предыдущий фрагмент демонстрирует, как мог бы выглядеть реактивный код при отсутствии API, поддерживающего составление цепочек из вызовов методов. Он наглядно показывает, что за кадром объекты `Flux*` включают друг в друга. После процесса сборки получаем цепочку издателей `Publisher`, в которой каждый последующий обортывает предыдущего. Следующий псевдокод демонстрирует это.

```
FluxFilter(
    FluxMap(
        FluxArray(1, 2, 3, 40, 500, 6000)
    )
)
```

Код наглядно показывает, как выглядит объект `Flux`, получившийся в результате применения последовательности операторов, такой как `just -> map -> filter`.

В жизненном цикле потока данных этот этап занимает важнейшее место, потому что во время сборки можно заменять операторы, проверяя тип потока данных. Например, последовательность операторов `concatWith -> concatWith -> concatWith` легко можно превратить в один вызов операции конкатенации. Вот как это делается в Reactor.

```
public final Flux<T> concatWith(Publisher<? extends T> other) {
    if (this instanceof FluxConcatArray) {
        @SuppressWarnings({ "unchecked" })
        FluxConcatArray<T> fluxConcatArray = (FluxConcatArray<T>) this;

        return fluxConcatArray.concatAdditionalSourceLast(other);
    }
}
```

```
    return concat(this, other);  
}
```

Как видим, если текущий экземпляр `Flux` относится к типу `FluxConcatArray`, то вместо `FluxConcatArray(FluxConcatArray(FluxA, FluxB), FluxC)` он создает один `FluxConcatArray(FluxA, FluxB, FluxC)` и тем самым улучшает общую производительность потока данных.

Кроме того, во время сборки можно подключить к потоку некоторые обработчики `Hooks` и включить расширенное журналирование, трассировку, сбор параметров или другие важные дополнения, которые могут пригодиться для отладки или мониторинга.

Подведем итог обсуждению этапа сборки в жизненном цикле реактивных потоков. В ходе этого этапа мы можем манипулировать конструкцией потока данных и применять разные приемы для оптимизации, мониторинга или отладки потока, которые являются неотъемлемой частью реактивной системы.

Этап подписки

Второй важный этап в жизненном цикле потока данных – подписка. Она происходит в момент, когда мы подписываемся на получение данных от заданного издателя `Publisher`. Например, следующий код осуществляет подписку на поток данных, который был собран в предыдущем разделе.

```
...  
filteredFlux.subscribe(...);
```

Как рассказывалось в предыдущих разделах, в процессе строительства конвейера обработки мы вкладываем издателей `Publisher` друг в друга и в результате получаем цепочку издателей. Подписываясь на самого внешнего издателя, запускаем процесс подписки в этой цепочке. Следующий псевдокод демонстрирует, как `Subscriber` распространяется по цепочке издателей на этапе подписки.

```
filterFlux.subscribe(Subscriber) {  
    mapFlux.subscribe(new FilterSubscriber(Subscriber)) {  
        arrayFlux.subscribe(new MapSubscriber(FilterSubscriber(Subscriber))) {  
            // здесь начинается передача фактических элементов  
        }  
    }  
}
```

Этот код показывает, что происходит на этапе подписки внутри собранного потока `Flux`. Как видите, вызов метода `filteredFlux.subscribe` последовательно вызывает метод `subscribe` каждого внутреннего издателя `Publisher`. Наконец, ког-

да выполнение достигнет строки с комментарием, мы будем иметь следующую последовательность подписчиков, обертывающих друг друга.

```
ArraySubscriber(
  MapSubscriber(
    FilterSubscriber(
      Subscriber
    )
  )
)
```

В отличие от цепочки экземпляров Flux, здесь мы имеем обертку ArraySubscriber, включающую пирамиду подписчиков Subscriber. В случае с пирамидой Flux в центре у нас был FluxArray (обращенная пирамида оберток).

Важность этапа подписки заключается в том, что на этой стадии можно выполнить те же оптимизации, что и на этапе сборки. Другой важный аспект – некоторые операторы, поддерживающие многопоточное выполнение в Reactor, позволяют изменить поток выполнения, в котором происходит подписка. Об оптимизации на этапе подписки и многопоточного выполнения мы расскажем далее в этой главе, а пока рассмотрим заключительный этап в жизненном цикле потоков данных.

Выполнение

Последний этап в жизненном цикле потока данных – выполнение. В ходе этого этапа происходит фактический обмен сигналами между издателем и подписчиком. Как мы знаем из стандарта Reactive Streams, первые два сигнала, которыми обмениваются издатель и подписчик, – это onSubscribe и request. Метод onSubscribe вызывается источником самого верхнего уровня, в данном случае ArrayPublisher. Он передает свой экземпляр Subscription указанному подписчику Subscriber. Ниже приводится псевдокод, иллюстрирующий процесс передачи подписки Subscription каждому подписчику Subscriber.

```
MapSubscriber(FilterSubscriber(Subscriber)).onSubscribe(
  new ArraySubscription()
) {
  FilterSubscriber(Subscriber).onSubscribe(
    new MapSubscription(ArraySubscription(...))
  ) {
    Subscriber.onSubscribe(
      FilterSubscription(MapSubscription(ArraySubscription(...)))
    ) {
      // здесь выполняется запрос данных
    }
  }
}
```

```
    }
}
```

После того как подписка будет передана всем подписчикам в цепочке и каждый подписчик упакует ее в конкретное представление, мы получим пирамиду подписок, представленную следующим кодом:

```
FilterSubscription(
  MapSubscription(
    ArraySubscription()
  )
)
```

Наконец, когда последний подписчик получит цепочку подписок, он должен вызвать метод `Subscription#request`, чтобы начать получать элементы. Следующий псевдокод показывает, как выглядит процесс запроса элементов.

```
FilterSubscription(MapSubscription(ArraySubscription(...)))
  .request(10) {
    MapSubscription(ArraySubscription(...))
      .request(10) {
        ArraySubscription(...)
          .request(10) {
            // начало отправки данных
          }
        }
      }
    }
```

Когда все подписчики передадут запрошенное количество элементов и `ArraySubscription` получит его, экземпляр `ArrayFlux` сможет начать передавать элементы в цепочку `MapSubscriber(FilterSubscriber(Subscriber))`. Следующий псевдокод описывает процесс передачи элементов подписчикам.

```
...
ArraySubscription.request(10) {
  MapSubscriber(FilterSubscriber(Subscriber)).onNext(1) {
    // здесь применяется отображение
    FilterSubscriber(Subscriber).onNext("1") {
      // элемент не соответствует фильтру,
      // запросить дополнительный элемент
      MapSubscription(ArraySubscription(...)).request(1) {...}
    }
  }
  MapSubscriber(FilterSubscriber(Subscriber)).onNext(20) {
    // здесь применяется отображение
    FilterSubscriber(Subscriber).onNext("20") {
      // элемент соответствует фильтру,
```

```
// передать его нижестоящему подписчику
Subscriber.onNext("20") {...}
}
}
```

Как видно из этого примера, в процессе выполнения элемент из источника проходит через цепочку подписчиков, и каждый выполняет определенные действия.

Важно понимать, что в процессе выполнения этого этапа можно применять оптимизации, сокращающие обмен сигналами. Например, как будет показано в следующих разделах, можно уменьшить число вызовов `Subscription#request` и тем самым увеличить производительность потока.



Как рассказывалось в главе 3 «*Reactive Streams – новый стандарт потоков*», метод `Subscription#request` может сохранять количество затребованных элементов в `volatile`-поле. Операция записи в такое поле обходится довольно дорого с вычислительной точки зрения, поэтому ее следует избегать, если это возможно.

В завершение знакомства с этапами жизненного цикла реактивного потока взгляните на рис. 4.8.

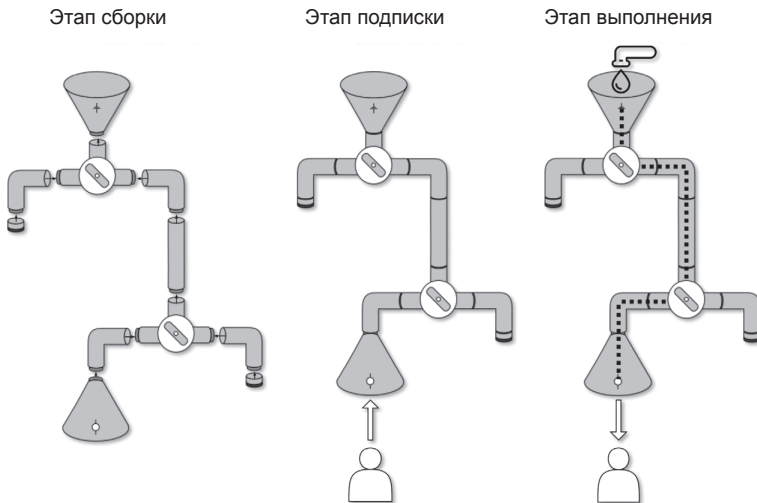


Рис. 4.8. Жизненный цикл реактивного потока

Итак, мы перечислили основные этапы жизненного цикла реактивных типов `Flux` и `Mono`. В следующих разделах расскажем, как Reactor обеспечивает высочайшую эффективность выполнения всех этих этапов в каждом реактивном потоке.

Модель планирования потоков выполнения в Reactor

В этом разделе познакомимся со средствами управления потоками выполнения из библиотеки Reactor и принципиальными отличиями между ними. Всего в библиотеке четыре оператора, позволяющих переключать рабочие потоки. Рассмотрим их по очереди.

Оператор `publishOn`

Оператор `publishOn` позволяет перенести **этап выполнения** потока данных в конкретный **рабочий поток**.



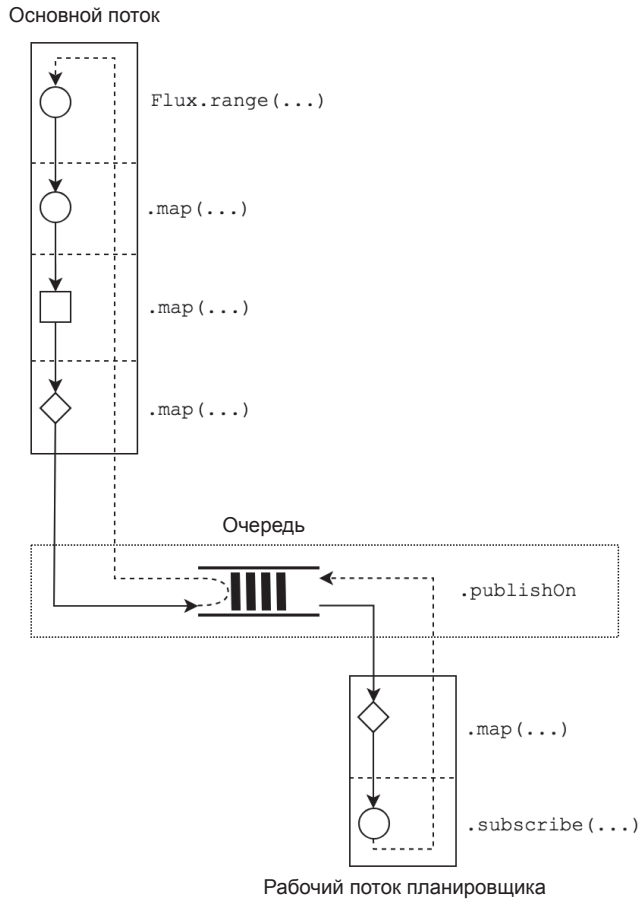
Мы намеренно используем термин «рабочий поток» вместо «поток выполнения», потому что фактический механизм планирования Scheduler может связывать рабочий поток с одним и тем же потоком выполнения, но с точки зрения самого планировщика задание будет выполняться разными *рабочими* потоками.

Чтобы задать рабочий поток для обработки элементов потока данных, Reactor вводит специальную абстракцию планировщика Scheduler. Scheduler – это интерфейс, представляющий рабочий поток или пул рабочих потоков в Project Reactor. Подробнее интерфейс Scheduler мы рассмотрим позже в этой же главе, а пока просто отметим, что он используется для выбора конкретного рабочего потока с целью обработки элементов потока данных. Чтобы понять, как использовать оператор `publishOn`, рассмотрим пример кода.

```
Scheduler scheduler = ...;           // (1)
//
Flux.range(0, 100)                   // (2) _|
    .map(String::valueOf)            // (3)  |> Главный поток выполнения
    .filter(s -> s.length() > 1)     // (4) _|
    .publishOn(scheduler)             // (5)

    .map(this::calculateHash)         // (6) _|
    .map(this::doBusinessLogic)       // (7)  |> Поток выполнения планировщика
    .subscribe()                     // (8) _|
```

Как видим, операции с элементами от шага 2 до шага 4 выполняются в главном потоке выполнения, а после вызова оператора `publishOn` – в другом, рабочем потоке планировщика Scheduler. Это означает, что `calculateHash` и `doBusinessLogic` выполняются в другом рабочем потоке выполнения, отличном от главного потока выполнения. Если взглянуть на оператор `publishOn` с точки зрения модели выполнения, можно заметить, что выполнение протекает так, как показано на рис. 4.9.

Рис. 4.9. Внутреннее устройство операторов `publishOn` из библиотеки Reactor

Как нетрудно заметить, оператор `publishOn` охватывает этап выполнения. Под капотом у оператора `publishOn` имеется очередь, куда записываются новые элементы, чтобы выделенный рабочий поток мог последовательно извлекать сообщения из очереди и обрабатывать их. В этом примере мы хотели показать, что работа выполняется в отдельном потоке, следовательно, этап выполнения имеет асинхронную границу. Как следствие теперь у нас есть два независимых этапа обработки потока данных. Важно подчеркнуть, что все элементы в реактивном потоке обрабатываются последовательно (не конкурентно), благодаря чему всегда можно определить точный порядок следования событий. Это свойство также называется **сериализуемостью**. Оно означает, что элемент, поступивший в `publishOn`, помещается в очередь, и как только наступит его черед, он будет извлечен из очереди и обработан. Обратите внимание, что обработка очереди осуществляется единственным рабочим потоком, поэтому порядок элементов всегда предсказуем.

Параллельная обработка с помощью publishOn

Прочитав предыдущий раздел, вы можете подумать, что оператор `publishOn` не поддерживает возможности конкурентной обработки реактивного потока. Однако это не так. Парадигма реактивного программирования, поддерживаемая библиотекой Project Reactor, допускает возможность параллельной обработки потока данных с использованием оператора `publishOn`. Например, рассмотрим сначала полностью синхронную обработку, изображенную на рис. 4.10.

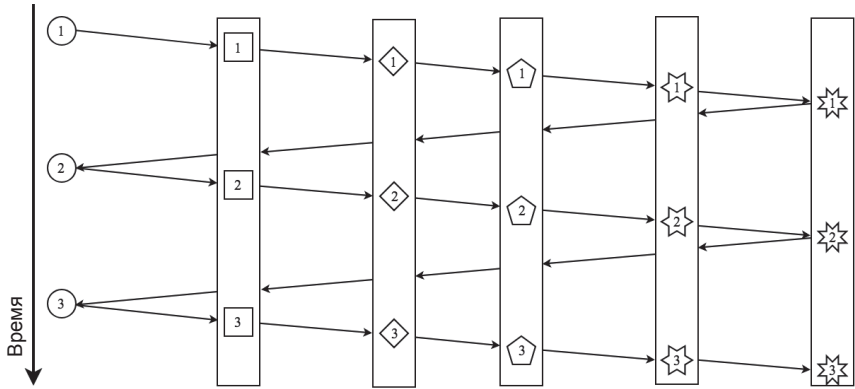


Рис. 4.10. Полностью синхронная обработка реактивного потока данных

Как показано на рис. 4.10, у нас есть поток данных с тремя элементами. Из-за естественных особенностей синхронной обработки элементов в потоке данных мы должны перемещать их один за другим через все этапы преобразования. Но, чтобы начать обработку следующего элемента, мы должны прежде закончить обработку предыдущего элемента. Напротив, если в этот процесс добавить `publishOn`, теоретически можно ускорить его выполнение. Диаграмма на рис. 4.11 демонстрирует тот же поток данных, но с включением оператора `publishOn`.

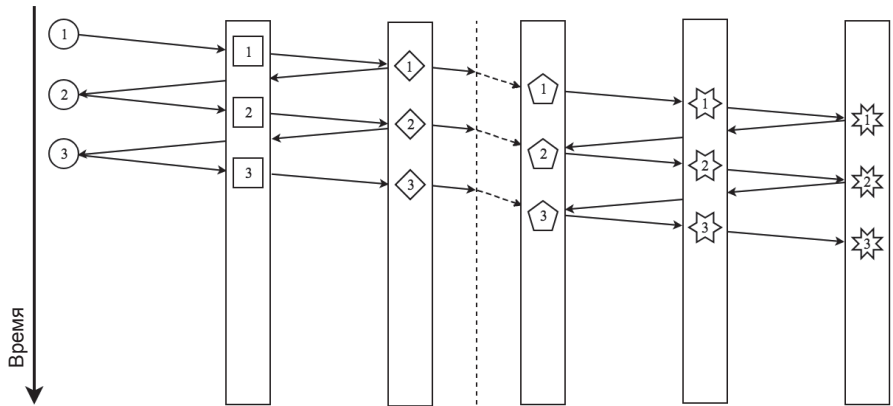


Рис. 4.11. Влияние оператора `publishOn` на обработку потока данных

Как показано на рис. 4.10, при неизменном времени обработки одного элемента просто за счет добавления асинхронной границы (введением оператора `publishOn`) между этапами обработки можно добиться параллельной обработки. Теперь левой стороне процесса обработки потока не требуется ждать завершения обработки элемента справа. Они могут работать независимо и параллельно.

Оператор `subscribeOn`

Другим важным фактором поддержки многопоточного выполнения в Reactor является оператор `subscribeOn`. В отличие от `publishOn`, оператор `subscribeOn` позволяет изменить рабочий поток выполнения, в котором выполняется часть цепочки, относящаяся к подписке. Этот оператор может пригодиться для создания источника данных, основанного на применении функции. Обычно в таких случаях данные генерируются на этапе подписки, поэтому и сами функции вызываются в момент вызова метода `.subscribe`. Рассмотрим следующий фрагмент кода, демонстрирующий, как можно организовать отправку данных с помощью `Mono.fromCallable`.

```
ObjectMapper objectMapper = ...
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
Mono.fromCallable(() ->
    objectMapper.readValue(json, Car.class)
)
...
```

Здесь `Mono.fromCallable` позволяет создать реактивный поток `Mono` из `Callable<T>` и передать результаты вычислений всем подписчикам `Subscriber`. Экземпляр `Callable` выполняется в момент вызова метода `.subscribe`, то есть `Mono.fromCallable` фактически делает следующее.

```
public void subscribe(Subscriber actual) {
    ...
    Subscription subscription = ...
    try {
        T t = callable.call();
        if (t == null) {
            subscription.onComplete();
        }
        else {
            subscription.onNext(t);
            subscription.onComplete();
        }
    }
    catch (Throwable e) {
        actual.onError(
            Operators.onOperatorError(e, actual.currentContext());
        );
    }
}
```

```
}
}
```

Как видно из предыдущего фрагмента, выполнение вызываемого объекта происходит в ходе выполнения метода `subscribe`. Это означает, что для смены рабочего потока выполнения, в котором происходит вызов `Callable`, нельзя использовать `publishOn`. К счастью, есть оператор `subscribeOn`, позволяющий определить, в каком рабочем потоке будет выполнен этап подписки. Следующий пример демонстрирует, как это можно реализовать.

```
Scheduler scheduler = ...;
Mono.fromCallable(...)
    .subscribeOn(scheduler)
    .subscribe();
```

Этот пример показывает, как выполнить заданный `Mono.fromCallable` в отдельном рабочем потоке выполнения. За кадром оператор `subscribeOn` выполняет подписку на поток данных `Publisher` внутри `Runnable`, который является планировщиком для заданного экземпляра `Scheduler`. Если сравнить модели выполнения операторов `subscribeOn` и `publishOn`, можно заметить следующее.



Рис. 4.12. Внутреннее устройство оператора `publishOn`

Как показано на рис. 4.12, оператор `subscribeOn` в какой-то мере может определять рабочий поток для этапа выполнения наряду с рабочим потоком для этапа подписки. Это возможно, потому что наряду с планированием выполнения метода `subscribe` он планирует каждый вызов метода `Subscription.request()`, то есть это происходит в рабочем потоке выполнения, который определит экземпляр `Scheduler`. Согласно стандарту `Reactive Streams`, издатель может начать отправку данных в поток выполнения вызывающего кода, поэтому последующий вызов `Subscriber.onNext()` произойдет в том же потоке выполнения, что и на-

чальный вызов `Subscription.request()`. Оператор `publishOn`, напротив, определяет поведение выполнения только для исходящего потока и не может влиять на выполнение входящего потока.

Оператор `parallel`

Кроме нетривиальных операторов для управления потоками выполнения, в которых должны протекать те или иные этапы обработки данных, библиотека Reactor предлагает уже знакомый способ распараллеливания работы. Для этого в Reactor имеется оператор `.parallel`, который позволяет разбить один поток данных на несколько подпотоков и распределить элементы между ними. Вот пример использования этого оператора.

```
Flux.range(0, 10000)
    .parallel()
    .runOn(Schedulers.parallel())
    .map()
    .filter()
    .subscribe()
```

Как видите, `.parallel()` является частью Flux API. Обратите особое внимание, что после применения оператора `parallel` мы начинаем работать с другим типом Flux – `ParallelFlux`. `ParallelFlux` – это абстракция группы экземпляров Flux, между которыми распределяются элементы из исходного потока Flux. Затем, применив оператор `runOn`, можно применить `publishOn` к внутренним экземплярам Flux и распределить работу по обработке элементов между несколькими рабочими потоками.

Планировщик

`Scheduler` – это интерфейс, определяющий два основных метода: `Scheduler.schedule` и `Scheduler.createWorker`. Первый позволяет запланировать задание `Runnable`, а второй возвращает выделенный экземпляр интерфейса `Worker`, который некоторым образом планирует задания `Runnable`. Главное отличие интерфейса `Scheduler` от интерфейса `Worker` в том, что `Scheduler` представляет пул рабочих потоков, тогда как `Worker` – выделенная абстракция потока выполнения `Thread` или некоторого ресурса. По умолчанию Reactor предлагает три основные реализации интерфейса `Scheduler`:

- `SingleScheduler` позволяет запланировать все возможные задания в одном выделенном рабочем потоке. Он поддерживает планирование с учетом времени, поэтому может использоваться для планирования периодических событий с задержкой. Сослаться на этот планировщик можно вызовом `Scheduler.single()`;

- `ParallelScheduler` работает с пулом рабочих потоков фиксированного размера (по умолчанию размер пула ограничивается числом ядер процессора). Хорошо подходит для вычислительных задач. Также способен планировать события во времени, например `Flux.interval(Duration.ofSeconds(1))`. Сослаться на этот планировщик можно вызовом `Scheduler.parallel()`;
- `ElasticScheduler` динамически создает рабочие потоки и кеширует пулы потоков выполнения. Максимальное число создаваемых пулов потоков выполнения не ограничивается, поэтому этот планировщик можно использовать для организации выполнения задач, связанных с вводом/выводом. Сослаться на этот планировщик можно вызовом `Scheduler.elastic()`.

Также можно определить свою реализацию `Scheduler` с требуемыми характеристиками. Пример создания своего планировщика для Reactor со средствами мониторинга приводится в главе 10 «И наконец, выпуск!».



Узнать больше об управлении потоками выполнения и планировщиках можно в документации к библиотеке Project (<http://projectreactor.io/docs/core/release/reference/#schedulers>).

Контекст

Еще одна важная особенность библиотеки Reactor – интерфейс `Context`. Экземпляры этого интерфейса передаются вместе с потоком данных. Основная идея интерфейса `Context` – дать доступ к некоторой контекстной информации, которая может пригодиться на этапе обработки. У кого-то из вас может возникнуть вопрос: «Зачем это нужно, если то же самое можно организовать с помощью `ThreadLocal`?» Например, многие фреймворки используют `ThreadLocal` для передачи `SecurityContext` в процессе обработки запроса пользователя и обеспечения доступа к информации об авторизованном пользователе из любой точки в конвейере. К сожалению, эта идея хороша, только когда используется механизм однопоточной обработки, когда обработка всегда связана с одним и тем же экземпляром `Thread`. Если использовать эту идею с асинхронной обработкой, `ThreadLocal` быстро утратит свою актуальность. Например, если выполнение будет протекать, как показано ниже, мы быстро потеряем `ThreadLocal`.

```
class ThreadLocalProblemShowcase {

    public static void main(String[] args) {
        ThreadLocal<Map<Object, Object>> threadLocal =           // (1)
            new ThreadLocal<>();                                  //
        threadLocal.set(new HashMap<>());                        // (1.1)

        Flux                                                     // (2)
            .range(0, 10)                                         // (2.1)
            .doOnNext(k ->                                         //
```

```

        threadLocal
            .get()
            .put(k, new Random(k).nextGaussian())
    )
    .publishOn(Schedulers.parallel())
    .map(k -> threadLocal.get().get(k))
    .blockLast();
}
}

```

Вот как действует этот код.

1. Объявляется экземпляр `ThreadLocal`. В строке (1.1) выполняется его подготовка для дальнейшего использования.
2. Объявляется поток данных `Flux`, который генерирует числовые элементы от 0 до 9 (2.1). Для каждого нового элемента в потоке генерируется вещественное значение с помощью `randomGaussian`, при этом элемент используется как начальное значение псевдослучайной последовательности. Затем сгенерированное значение помещается в ассоциативный массив `ThreadLocal`. В строке (2.3) выполнение передается в другой поток `Thread`. Наконец, в строке (2.4) число из потока отображается в случайное значение, сгенерированное ранее и сохраненное в ассоциативном массиве `ThreadLocal`. В этот момент мы получим `NullPointerException`, потому что ассоциативный массив, созданный в главном потоке выполнения, недоступен в другом потоке выполнения.

Как видим, использование `ThreadLocal` в многопоточном окружении очень опасно и может стать причиной непредсказуемого поведения. Даже притом что Java API позволяет передавать `ThreadLocal` из одного потока выполнения в другой, он не гарантирует абсолютную надежность такой передачи.

К счастью, интерфейс `Context` из библиотеки `Reactor` решает эту проблему.

```

Flux.range(0, 10)
    .flatMap(k ->
        Mono.subscriberContext()
            .doOnNext(context -> {
                Map<Object, Object> map = context.get("randoms");
                map.put(k, new Random(k).nextGaussian());
            })
            .thenReturn(k)
    )
    .publishOn(Schedulers.parallel())
    .flatMap(k ->
        Mono.subscriberContext()
            .map(context -> {
                Map<Object, Object> map = context.get("randoms");
            })
    )

```

```
        return map.get(k);                                // (2.2)
    })                                                    //
)                                                        //
.subscriberContext(context ->                           // (3)
    context.put("randoms", new HashMap()))              //
)                                                        //
.blockLast();                                           //
```

Вот как действует этот код.

1. Данная строка демонстрирует обращение к реализации Context в Reactor. Как видите, доступ к экземпляру Context в текущем потоке данных осуществляется вызовом статического оператора `subscriberContext`. Так же как в предыдущем примере, получив экземпляр Context (1.1), мы можем обратиться к хранимому экземпляру Map (1.2) и сохранить сгенерированное значение. Наконец, возвращаем начальный параметр `flatMap`.
2. Снова обращаемся к экземпляру Context, уже после переключения потока выполнения. Даже притом что этот фрагмент идентичен соответствующему фрагменту в предыдущем примере, где мы использовали `ThreadLocal`, в строке (2.1) мы благополучно извлекаем хранимый ассоциативный массив и получаем сгенерированное случайное значение (2.2).
3. В заключение добавляем новый ассоциативный массив "randoms" во входящий поток в составе нового экземпляра Context.

Из предыдущего примера можно сделать следующие выводы: экземпляр Context доступен через оператора `Mono.subscriberContext` и может добавляться в поток данных с использованием оператора `subscriberContext(Context)`.

При изучении предыдущего примера может возникнуть вопрос, почему для передачи данных обязательно использовать Map, если интерфейс Context имеет тот же метод, который имеет и интерфейс Map. Дело в том, что Context изначально проектировался как неизменяемый объект и после добавления в него нового элемента мы получим новый экземпляр Context. Такое решение было принято для поддержки многопоточной модели доступа. Это единственный способ передать Context с потоком данных и иметь возможность динамически добавлять в него информацию, которая должна быть доступна в течение всего этапа сборки или подписки. В данном случае, если Context будет создан на этапе сборки, все подписчики смогут использовать один и тот же статический контекст, но это может быть неудобно в ситуациях, когда каждый подписчик должен иметь свой контекст (например, представляющий соединение с пользователем). Поэтому единственный этап в жизненном цикле, когда подписчик Subscriber может определить свой контекст, – это этап подписки.

Как рассказывалось в предыдущих разделах, на этапе подписки подписчик поднимается снизу вверх по цепочке издателей и на каждом этапе оборачивается

в локальное представление `Subscriber`, предлагающее дополнительную логику обработки. Чтобы сохранить этот процесс неизменным и обеспечить передачу дополнительного объекта `Context`, `Reactor` использует расширенную версию интерфейса `Subscriber` – интерфейс `CoreSubscriber`. Он позволяет передавать `Context` как поле. Ниже показано, как выглядит интерфейс `CoreSubscriber`.

```
interface CoreSubscriber<T> extends Subscriber<T> {  
    default Context currentContext() {  
        return Context.empty();  
    }  
}
```

Как можно заметить в этом фрагменте, `CoreSubscriber` вводит дополнительный метод с именем `currentContext`. Он обеспечивает доступ к текущему объекту `Context`. Большинство операторов в `Project Reactor` предлагает реализацию интерфейса `CoreSubscriber` со ссылкой на `Context` в исходящем потоке. Как вы наверняка заметили, единственный оператор, с помощью которого можно изменить текущий объект `Context`, – это `subscriberContext`, возвращающий реализацию `CoreSubscriber`, которая хранит контекст исходящего потока и передается как параметр.

Кроме того, такое поведение означает, что доступный объект `Context` может отличаться в разных точках в потоке данных. Это поведение, например, демонстрирует следующий фрагмент.

```
void run() {  
    printCurrentContext("top")  
    .subscriberContext(Context.of("top", "context"))  
    .flatMap(__ -> printCurrentContext("middle"))  
    .subscriberContext(Context.of("middle", "context"))  
    .flatMap(__ -> printCurrentContext("bottom"))  
    .subscriberContext(Context.of("bottom", "context"))  
    .flatMap(__ -> printCurrentContext("initial"))  
    .block();  
}  
  
void print(String id, Context context) {  
    ...  
}  
  
Mono<Context> printCurrentContext(String id) {  
    return Mono  
        .subscriberContext()  
        .doOnNext(context -> print(id, context));  
}
```

Мы видим, как можно использовать Context на этапе конструирования потока. Если выполнить предыдущий код, он выведет следующие строки:

```
top {  
    Context3{bottom=context, middle=context, top=context}  
}  
  
middle {  
    Context2{bottom=context, middle=context}  
}  
  
bottom {  
    Context1{bottom=context}  
}  
  
initial {  
    Context0{}}
```

Как видите, объект Context, доступный на вершине потока, содержит полный контекст, доступный в этом потоке, тогда как в середине доступен только объект Context, который был определен ниже в потоке, а контекст потребителя в самом низу (с идентификатором initial) вообще имеет пустой Context.

В общем и целом поддержка контекста – потрясающая особенность, поднимающая Project Reactor на следующий уровень инструментов создания реактивных систем. Кроме того, эта особенность с успехом может использоваться с самыми разными целями, когда требуется доступ к некоторым контекстным данным, например в середине процесса обработки пользовательского запроса. Как увидим в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux», эта особенность широко используется в Spring Framework, особенно в реактивной библиотеке Spring Security.

Несмотря на довольно подробный охват поддержки контекста, существует еще масса возможностей его использования вместе с другими приемами Reactor. Чтобы узнать больше, обращайтесь к разделу документации в библиотеке Project Reactor, доступному по ссылке <http://projectreactor.io/docs/core/release/reference/#context>.

Особенности внутренней реализации Project Reactor

Как мы видели в предыдущем разделе, библиотека Reactor обладает богатым набором полезных операторов. Кроме того, можно было заметить, что в целом API библиотеки имеет операторы, добавляющие сходство с RxJava. Но в чем глав-

ное отличие между библиотеками старого и нового поколений, включая Project Reactor 3? Какое достижение является самым важным? Одно из самых заметных усовершенствований – это **жизненный цикл реактивного потока** и **слияние операторов**. В предыдущем разделе мы рассмотрели жизненный цикл реактивного потока, а теперь исследуем слияние операторов.

Макрослияние

Макрослияние (macro-fusion) происходит в основном на этапе сборки. Его цель – заменить один оператор другим. Например, мы уже знаем, что тип Mono оптимизирован для обработки только одного элемента. В то же время некоторые части операторов внутри типа Flux также предполагают обработку одного или нуля элементов (например, операторы just(T), empty() и error(Throwable)). В большинстве случаев эти простые операторы используются совместно с другими потоками преобразования. Следовательно, крайне важно максимально сократить накладные расходы. С этой целью библиотека Reactor осуществляет оптимизацию на этапе сборки, и если обнаруживается, что вышестоящий издатель Publisher реализует такие интерфейсы, как Callable или ScalarCallable, он будет заменен оптимизированным оператором. Примером применения такой оптимизации может служить следующий код:

```
Flux.just(1)
    .publishOn(...)
    .map(...)
```

Предыдущий код демонстрирует действительно простой пример, когда обработка элемента должна передаваться для обработки в другой поток выполнения сразу после его создания. Без оптимизации такой код создаст очередь для хранения элементов из другого рабочего потока выполнения, плюс добавление элементов в очередь и их извлечение из очереди сопряжено с выполнением операций чтения/записи из памяти, общей для нескольких потоков выполнения, поэтому такой простой код приносит существенные накладные расходы. К счастью, этот процесс можно оптимизировать. Так как не важно, в каком рабочем потоке выполнения произойдет обработка, и передачу одного элемента можно представить как ScalarCallable#call, можем заменить оператор publishOn на subscribeOn, который не требует создания дополнительной очереди. Более того, обработка исходящего потока данных от этой оптимизации не изменится, поэтому оптимизированный поток даст нам тот же результат.

Предыдущий пример – это пример оптимизации макрослияния, скрытого в Project Reactor. В разделе, посвященном этапу сборки потока (выше в данной главе), упоминался другой пример такой оптимизации. Вообще, целью оптимизации макрослияния в Project Reactor является оптимизация этапа сборки, чтобы дать возможность для стрельбы по воробьям вместо пушки использовать более простое и дешевое оружие.

Микрослияние

Микрослияние (micro-fusion) – более сложная оптимизация. Она связана с оптимизацией этапа выполнения и повторным использованием ресурсов. Хорошим примером микрослияния может служить условный оператор. Чтобы понять суть, рассмотрим диаграмму на рис. 4.13.



Рис. 4.13. Проблема проверки качества товара на примере грузовика

Представьте следующую ситуацию. Магазин заказал доставку n единиц товара. Спустя некоторое время фабрика отправила товар на грузовике в магазин. Однако, прежде чем прибыть в магазин, грузовик должен заехать в отдел контроля, где проверяют качество товара. К сожалению, некоторые предметы были упакованы небрежно, и только часть из них доставили в магазин. После этого фабрика загрузила еще один грузовик и снова отправила товар в магазин. Так повторялось несколько раз, пока весь товар не доставили в магазин. К счастью, на фабрике поняли, что потратили слишком много времени и денег на доставку товара через отдел контроля, и решили создать свой внутренний отдел контроля, наняв специалиста из внешнего отдела контроля (рис. 4.14).



Рис. 4.14. Решение проблемы проверки качества товара с привлечением специалиста на стороне фабрики

Теперь весь товар проверяется прямо на фабрике и затем отправляется в магазин без заезда в отдел контроля.

Как эта история связана с программированием? Рассмотрим следующий пример.

```
Flux.from(factory)
    .filter(inspectionDepartment)
    .subscribe(store);
```


Здесь мы имеем похожую ситуацию. Подписчик запрашивает определенное количество элементов из источника. Производимые элементы, двигаясь сквозь цепочку операторов, проходят через условный оператор `filter`, который может «забраковать» некоторые из них. Чтобы удовлетворить требование подписчика, для каждого отфильтрованного элемента оператор `filter` должен послать издателю дополнительный запрос `request(1)`. В современных реактивных библиотеках (таких как RxJava или Reactor 3) операция запроса добавляет свои накладные расходы.



Согласно исследованиям Дэвида Карнока (David Karnok), каждый «вызов `request()`» обычно влечет выполнение атомарного цикла сравнения с обменом (Compare and Swap, CAS) длительностью 21–45 тактов на каждый элемент».

Это означает, что условные операторы, такие как `filter`, могут оказывать серьезное влияние на общую производительность! По данной причине в свое время был создан тип `ConditionalSubscriber`. Он позволяет организовать проверку условия непосредственно на стороне источника и передачу необходимого количества элементов без дополнительных вызовов `request`.

Вторая оптимизация микрослияния считается самой сложной. Она имеет отношение к асинхронным границам между операторами, упоминавшимся в главе 3 «*Reactive Streams – новый стандарт потоков*». Чтобы понять суть, представьте цепочку операторов с несколькими асинхронными границами, как в следующем примере.

```
Flux.just(1, 2, 3)
    .publishOn(Schedulers.parallel())           // (1)
    .concatMap(i -> Flux.range(0, i)
        .publishOn(Schedulers.parallel()))      // (2)
    .subscribe();
```

Это цепочка операторов из библиотеки Reactor. Она включает две асинхронные границы, следовательно, в этих местах появятся очереди. Например, по своей природе оператор `concatMap` способен генерировать несколько новых элементов для каждого входящего. По этой причине невозможно предсказать, сколько элементов будет произведено внутренними экземплярами Flux. Чтобы правильно обработать обратное давление и избежать избыточной нагрузки на потребителя, результаты лучше помещать в очередь. Оператор `publishOn` также нуждается во внутренней очереди для передачи элементов реактивного потока данных из одного рабочего потока выполнения в другой. Кроме издержек на создание и обслуживание очередей здесь также могут выполняться дорогостоящие вызовы `request()` через асинхронные границы. Они могут повлечь еще более существенные затраты памяти. Чтобы было понятнее, рассмотрим диаграмму на рис. 4.15.

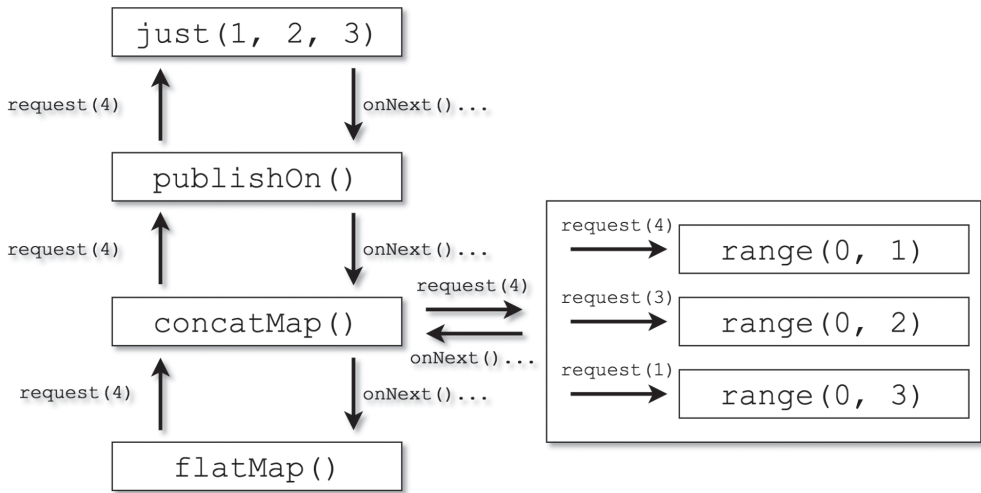


Рис. 4.15. Затраты на асинхронные границы без оптимизации

Диаграмма на рис. 4.15 отражает внутреннее поведение предыдущего фрагмента кода. Здесь мы имеем огромные накладные расходы внутри `concatMap`, где приходится вызывать `request` для каждого внутреннего потока данных, чтобы удовлетворить требование подписчика. Любой оператор с очередью имеет свой цикл CAS, который при использовании неправильной модели запросов может существенно влиять на общую производительность. Например, вызов `request(1)` или с любым другим числом элементов, которое выглядит незначительным с общим количеством данных, можно смело считать неправильной моделью запросов.



Цикл сравнения с обменом (*Compare and Swap, CAS*) – это атомарная операция, которая возвращает значение 1 или 0 в зависимости от результата. Так как нам требуется добиться успеха, мы повторяем операции CAS снова и снова, пока они не преуспеют. Такие повторяемые операции CAS называют *циклом CAS*.

Чтобы предотвратить потери памяти и производительности, протоколы взаимодействия следует переключать так, как рекомендуется стандартом Reactive Streams. Допустим, в пределах границ имеется цепочка элементов, сохраняемых в общей очереди, тогда замена всей цепочки операторов одним оператором выше в потоке, выполняющим функцию очереди без дополнительных вызовов `request`, может значительно улучшить общую производительность. При таком подходе подписчик может извлекать значения из потока, пока тот не вернет `null`, чтобы показать, что достигнут конец потока данных. Также в виде исключения для уведомления о доступности элементов можно вызывать `onNext` подписчика со значениями `null`. Кроме того, об ошибках или о завершении потока данных можно уведомлять как обычно – с помощью `onError` и `onComplete`. При таком подходе предыдущий пример можно оптимизировать, как показано на рис. 4.16.

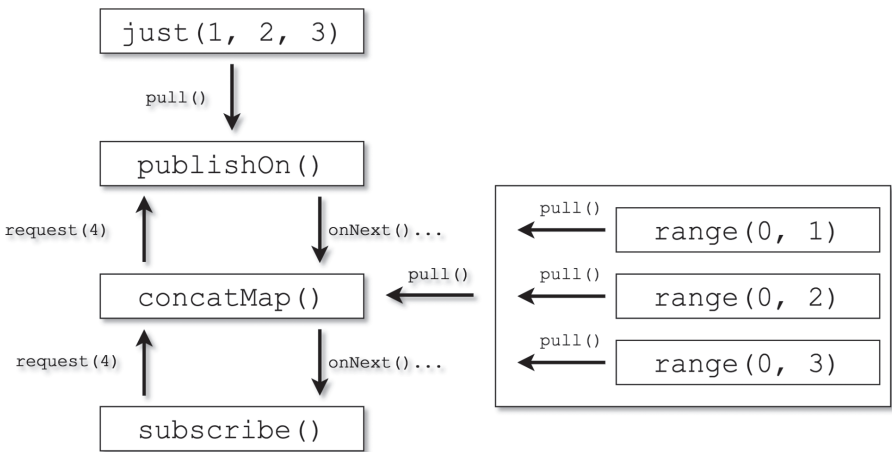


Рис. 4.16. Очередь подписки и переключение протокола

В этом примере операторы `publishOn` и `concatMap` можно существенно оптимизировать. В первом случае отсутствуют промежуточные операторы, которые должны выполняться в главном потоке выполнения. То есть вместо очереди можно использовать непосредственно оператор `just` и извлекать из него данные в отдельном потоке выполнения. В случае с `concatMap` все внутренние потоки данных тоже можно рассматривать как очереди, каждая из которых будет опустошаться без дополнительных вызовов `request`.



Следует отметить, что ничто не мешает наладить взаимодействие между `publishOn` и `concatMap`, используя оптимизированный протокол, но на момент написания этих строк такая оптимизация еще не была реализована, поэтому мы решили показать механизмы как есть.

Итак, как вы могли убедиться, изнутри библиотека Reactor выглядит намного сложнее. Благодаря мощным оптимизациям Reactor намного опережает RxJava 1.x и обеспечивает лучшую производительность.

Заключение

В этой главе мы охватили широкий спектр тем. Кратко рассмотрели историю развития Reactor и выяснили мотивы, подтолкнувшие разработчиков к созданию *еще одной реактивной библиотеки* – Project Reactor. Познакомились с наиболее важными этапами, которые прошли разработчики, чтобы создать такой гибкий и универсальный инструмент. Исследовали основные проблемы реализации RxJava 1.x, а также ранних версий Reactor. Затем, рассмотрев изменения в Project Reactor после выхода стандарта Reactive Streams, мы объяснили, почему такое простое и эффективное реактивное программирование требует такой сложной реализации.

Также мы описали реактивные типы Mono и Flux и разные способы создания, преобразования и получения реактивных потоков. Мы заглянули внутрь действующего потока данных и показали, как управлять обратным давлением с использованием модели *PULL/PUSH* объекта Subscription. Рассказали, как слияние операторов способствует увеличению производительности реактивных потоков. Подводя итог, скажем, что библиотека Project Reactor предлагает набор мощных инструментов для реактивного программирования, а также для создания асинхронных приложений и приложений, интенсивно выполняющих операции ввода/вывода.

В следующих главах расскажем, какие усовершенствования были внесены в Spring Framework, чтобы добавить в него поддержку реактивного программирования в целом и Project Reactor в частности. Там сосредоточимся на проблемах разработки приложений с использованием Spring 5 WebFlux и Reactive Spring Data.

Глава 5

Добавление реактивности с помощью Spring Boot 2

В предыдущей главе мы познакомились с основами Project Reactor, исследовали поведение реактивных типов и операторов и узнали, как они помогают решать разные бизнес-задачи. Мы выяснили, что под простым API спрятаны сложные механизмы конкурентной, асинхронной и неблокирующей обработки сообщений. Дополнительно исследовали методы управления обратным давлением и прочие родственные стратегии. Как вы могли заметить в предыдущей главе, Project Reactor – это больше, чем библиотека поддержки реактивного программирования. Она также предлагает дополнения и адаптеры, позволяющие создавать реактивные системы даже без Spring Framework. Благодаря этому мы узнали о возможности интеграции Project Reactor с Apache Kafka и Netty.

Библиотека Project Reactor с успехом может использоваться без Spring Framework, но обычно этого недостаточно для создания многофункциональных приложений. Одним из недостающих элементов в данном случае является широко известный механизм внедрения зависимостей, помогающий отделить компоненты друг от друга. Кроме того, чем более сложные и мощные приложения мы создаем, тем ярче блистает Spring Framework. Однако еще лучше приложения получаются, когда для их создания используется Spring Boot.

По этой причине в данной главе поговорим о важности библиотеки Spring Boot и о возможностях, которые она открывает. Познакомимся также с грядущими изменениями в Spring Framework 5 и Spring Boot 2 и посмотрим, как экосистема Spring берет на вооружение приемы реактивного программирования.

Будут рассмотрены следующие темы:

- какие проблемы решает Spring Boot и как;
- основы Spring Boot;
- реактивность в Spring Boot 2.0 и Spring Framework.

Быстрый старт как ключ к успеху

Люди никогда не любили тратить много времени на рутинную работу и решение задач, не связанных с основной целью. В бизнесе, чтобы достичь желаемого, нужно быстро учиться и быстро экспериментировать. То же верно и для реактивного программирования. Для нас очень важно быстро реагировать на изменения на рынке, быстро менять стратегии и быстро достигать поставленных целей. Чем быстрее докажем ценность своей идеи, тем скорее она будет внедрена в бизнес и тем меньше денег потратим на исследования.

Люди всегда стремятся упростить рутинную работу, и разработчик не исключение. Мы любим, когда все работает «из коробки», особенно если речь заходит о чем-то сложном, таком как фреймворк Spring. Несмотря на множество преимуществ и наличие полезных функций в Spring Framework, требуется глубокое понимание особенностей работы с ним, и начинающие разработчики легко могут потерпеть неудачу, попав в область, где они не имеют опыта. Хорошим примером может послужить настройка контейнера с инверсией управления (**Inversion of Control, IoC**), допускающая по меньшей мере пять возможных способов. Чтобы понять суть проблемы, рассмотрим следующий фрагмент кода:

```
public class SpringApp {
    public static void main(String[] args) {
        GenericApplicationContext context =
            new GenericApplicationContext();

        new XmlBeanDefinitionReader(context)
            .loadBeanDefinitions("services.xml");

        new GroovyBeanDefinitionReader(context)
            .loadBeanDefinitions("services.groovy");

        new PropertiesBeanDefinitionReader(context)
            .loadBeanDefinitions("services.properties");

        context.refresh();
    }
}
```

Как видим, сам Spring Framework поддерживает как минимум три разных способа регистрации компонентов в контексте Spring.

С одной стороны, Spring Framework предлагает гибкие настройки источников компонентов. С другой – такой обширный список возможностей порождает ряд проблем. Например, одна из проблем – использование для описания конфигурации разметки XML, которая *не имеет простого способа отладки*. Другая проблема, усложняющая работу с конфигурациями, – *невозможность убедиться в их правиль-*

ности без дополнительных инструментов, таких как IntelliJ IDEA или Spring Tool Suite. Наконец, отсутствие надлежащей дисциплины в оформлении кода и соглашений может существенно усложнить большие проекты и снизить их ясность. Сложно, если один разработчик определяет компоненты в XML, а другой – в свойствах. Кто-то третий может делать то же самое в конфигурациях Java. Как следствие новый разработчик легко запутается в этой несогласованности и будет вникать в проект намного дольше, чем необходимо.

Кроме простого механизма IoC Spring Framework предлагает более сложные средства, такие как модули Spring Web и Spring Data. Оба модуля требуют большого количества настроек, только чтобы запустить приложение. Проблемы обычно возникают, когда разрабатываемое приложение зависит от платформы, на которой запускается, а значит, требует увеличенного объема настроек и шаблонного кода.



Обратите внимание, что *независимость от платформы* означает независимость от конкретного серверного API, такого как Servlet API. Сюда же можно отнести независимость от конкретного окружения и других особенностей.

Например, для настройки элементарного веб-приложения требуется не менее семи строк шаблонного кода, как показано ниже.

```
public class MyWebApplicationInitializer
    implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletCxt) {
        AnnotationConfigWebApplicationContext cxt =
            new AnnotationConfigWebApplicationContext();
        cxt.register(AppConfig.class);
        cxt.refresh();
        DispatcherServlet servlet = new DispatcherServlet(cxt);
        ServletRegistration.Dynamic registration = servletCxt
            .addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

В этот код не входят никакие настройки безопасности или другие важные функции, такие как отображение контента. В какой-то момент каждое приложение на основе Spring получает похожие фрагменты кода, совершенно неоптимизированные и требующие дополнительного внимания со стороны разработчиков, что влечет за собой напрасную трату денег.

Использование Spring Roo для ускорения разработки приложений

К счастью, разработчики Spring осознают важность быстрого запуска проектов. В начале 2009 года был создан новый проект под названием Spring Roo (подробности ищите по ссылке <https://projects.spring.io/spring-roo>). Цель проекта – ускорить разработку приложений. Основная идея Spring Roo заключается в использовании парадигмы **преимущества соглашений перед конфигурацией**. Для этого Spring Roo предлагает интерфейс командной строки, который позволяет инициализировать инфраструктуру и предметные модели и создавать REST API несколькими командами. Spring Roo упрощает процесс разработки приложений. Однако этот инструмент, по всей видимости, не получится использовать в больших приложениях. На практике часто возникали проблемы, когда сложность структуры проекта преодолевала некоторый порог или когда использовались технологии, не входящие в состав Spring Framework. Наконец, сам Spring Roo не получил большой популярности. Следовательно, проблема быстрой разработки остается открытой.



Обратите внимание, что на момент написания этих строк вышла версия Spring Roo 2.0. Она содержит множество улучшений, с которыми можно ознакомиться здесь: <https://docs.spring.io/spring-roo/docs/current/reference/html>.

Spring Boot как ключ к созданию быстро растущих приложений

В конце 2012 года Майк Янгстрем (Mike Youngstrom) поднял проблему, которая повлияла на будущее Spring Framework. Он предложил изменить всю архитектуру Spring и упростить использование Spring Framework, чтобы разработчики могли начинать создавать бизнес-логику как можно быстрее. Несмотря на то что это предложение было отклонено, оно побудило разработчиков Spring создать новый проект, существенно упрощающий использование Spring Framework. В середине 2013 года вышла предварительная версия проекта с названием Spring Boot (подробности ищите по ссылке <https://spring.io/projects/spring-boot>). Главная идея Spring Boot заключалась в том, чтобы упростить процесс разработки приложений и дать пользователям возможность начинать новые проекты без дополнительных настроек инфраструктуры.

Наряду с этим Spring Boot воплощает идею бесконтейнерного веб-приложения и использования «толстых» (fat) JAR-файлов. В соответствии с этим подходом приложение на основе Spring можно уместить в строку кода и запускать его од-

ной дополнительной командой. В следующем фрагменте демонстрируется законченное веб-приложение на Spring Boot.

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Самой важной частью в предыдущем примере является аннотация `@SpringBootApplication`, необходимая для запуска контейнера IoC. Также существует сервер MVC и другие компоненты приложений. Давайте рассмотрим этот проект подробнее. Прежде всего Spring Boot – это набор модулей и дополнений к современным инструментам сборки, таким как Gradle и Maven. Вообще говоря, Spring Boot имеет два основных модуля. Первый из них – модуль `spring-boot`, в который входят все возможные конфигурации по умолчанию, имеющие отношение к контейнеру Spring IoC. Второй – модуль `spring-boot-autoconfigure`, включающий все возможные конфигурации для всех имеющихся проектов Spring, таких как Spring Data, Spring MVC, Spring WebFlux и др.

На первый взгляд кажется, что все имеющиеся конфигурации включены одновременно, даже если они не требуются. На самом деле это не так, и все конфигурации отключены до момента появления соответствующей зависимости. Spring Boot определяет новое понятие для модулей, содержащих в своих именах слово `-starter-`. По умолчанию такие модули не содержат кода на Java, но включают все зависимости, необходимые для активации конкретных конфигураций в `spring-boot-autoconfigure`. Благодаря Spring Boot мы теперь имеем модули `-starter-web` и `-starter-data-jpa`, помогающие без лишних хлопот настраивать все элементы инфраструктуры. Самым заметным отличием проекта Spring Roo является большая гибкость. Наряду со стандартными конфигурациями, которые легко расширяются, Spring Boot предлагает гибкий API для создания своих модулей `-starter-`. Этот API позволяет заменять стандартные конфигурации и использовать наши собственные конфигурации для конкретных модулей.



В этой книге мы не будем углубляться в детали Spring Boot. Однако желающие могут прочитать книгу Грегга Л. Турнквиста (Greg L. Turnquist) «*Preview Online Code Files Learning Spring Boot 2.0, Second Edition*», где очень подробно исследуется Spring Boot. Она доступна по ссылке <https://www.packtpub.com/application-development/learning-spring-boot-20-second-edition>.

Реактивность в Spring Boot 2.0

В книге рассказывается о реактивном программировании, поэтому не будем слишком вдаваться в детали Spring Boot. Однако, как говорилось в предыдущем

разделе, возможность быстро начать работу над приложением является важнейшим ингредиентом успеха фреймворка. Давайте разберемся, как реактивность отразилась на экосистеме Spring. Поскольку простая смена парадигмы программирования не дает никакой выгоды из-за блокирующей природы модулей Spring MVC и Spring Data, разработчики Spring решили сменить парадигму также внутри этих модулей. С этой целью экосистема Spring предлагает целый список реактивных модулей. В данном разделе мы кратко опишем каждый из них. Однако некоторые из модулей будут более подробно описаны далее в отдельных главах.

Реактивность в Spring Core

Spring Core – центральный модуль экосистемы Spring. Одним из примечательных улучшений в Spring Framework 5.x стала встроенная поддержка реактивных потоков и реактивных библиотек, таких как RxJava 1/2 и Project Reactor 3.

Поддержка преобразования реактивных типов

Одним из самых значительных улучшений в поддержке стандарта Reactive Streams стало появление классов `ReactiveAdapter` и `ReactiveAdapterRegistry`. Класс `ReactiveAdapter` предлагает два основных метода преобразования реактивных типов, как показано в следующем фрагменте:

```
class ReactiveAdapter {
    ...
    <T> Publisher<T> toPublisher(@Nullable Object source) { ... } // (1)
    Object fromPublisher(Publisher<?> publisher) { ... } // (2)
}
```

Эти два метода предназначены для преобразования любых типов в тип `Publisher<T>` (1) и обратно в тип `Object`. Например, чтобы организовать преобразование реактивного типа `Maybe` из библиотеки RxJava 2, можно создать свою версию `ReactiveAdapter`.

```
public class MaybeReactiveAdapter extends ReactiveAdapter { // (1)
    public MaybeReactiveAdapter() { // (2)
        super(
            ReactiveTypeDescriptor // (3)
                .singleOptionalValue(Maybe.class, Maybe::empty), //
            rawMaybe -> ((Maybe<?>)rawMaybe).toFlowable(), // (4)
            publisher -> Flowable.fromPublisher(publisher) // (5)
                .singleElement() //
        );
    }
}
```

В этом примере мы унаследовали класс `ReactiveAdapter` и представили свою реализацию (1). Мы создали свой конструктор по умолчанию и скрыли в нем детали реализации (2). Первый параметр в вызове родительского конструктора (3) – определение экземпляра `ReactiveTypeDescriptor`.

`ReactiveTypeDescriptor` хранит информацию о реактивном типе, использованном в `ReactiveAdapter`. Наконец, родительский конструктор требует определить функции (в данном случае это лямбда-выражения), преобразующие исходный объект (в данном случае типа `Maybe`) в экземпляры `Publisher` (4) и обратно.



Обратите внимание: `ReactiveAdapter` предполагает, что перед передачей объекта методу `toPublisher` совместимость типа объекта будет проверена с помощью метода `ReactiveAdapter#getReactiveType`.

Для большей простоты существует `ReactiveAdapterRegistry`, позволяющий хранить экземпляры `ReactiveAdapter` в одном месте и обобщающий доступ к ним, как показано ниже:

```
ReactiveAdapterRegistry
    .getSharedInstance()                // (1)
    .registerReactiveType(              // (2)
        ReactiveTypeDescriptor
            .singleOptionalValue(Maybe.class, Maybe::empty),
        rawMaybe -> ((Maybe<?>)rawMaybe).toFlowable(),
        publisher -> Flowable.fromPublisher(publisher)
            .singleElement()
    );
...
ReactiveAdapter maybeAdapter = ReactiveAdapterRegistry
    .getSharedInstance()                // (1)
    .getAdapter(Maybe.class);           // (3)
```

Как видите, `ReactiveAdapterRegistry` представляет общий пул экземпляров `ReactiveAdapter` для разных реактивных типов. Он позволяет создать только один экземпляр (1) каждого адаптера, который может использоваться во многих местах внутри фреймворка или разрабатываемого приложения. Наряду с этим реестр позволяет зарегистрировать адаптер, передав те же входные параметры, что и в предыдущем примере (2). Наконец, есть возможность получить имеющийся адаптер, передав Java-класс, который требуется преобразовать (3).

Реактивный ввод/вывод

Другим существенным улучшением, связанным с поддержкой реактивности, стало усовершенствование основных функций ввода/вывода. Прежде всего в модуль `Spring Core` добавили дополнительную абстракцию над буфером байтов, которая получила имя `DataBuffer`. Основной причиной этого шага было стремление отказаться от `java.nio.ByteBuffer` и предложить абстракцию, способную дать под-

держку разных буферов байтов без необходимости выполнять любые преобразования между ними. Например, чтобы преобразовать `io.netty.buffer.ByteBuf` в `ByteBuffer`, необходим доступ к хранимым байтам, для чего может понадобиться прочитывать их из памяти за пределами кучи и записать в кучу. Это может нанести серьезный ущерб эффективности использования памяти и помешать повторному использованию буфера, предлагаемого библиотекой Netty. Для таких случаев `Spring DataBuffer` предлагает абстракцию конкретной реализации и позволяет использовать базовые реализации обобщенным способом. Дополнительный интерфейс `PooledDataBuffer` также поддерживает механизм подсчета ссылок и обеспечивает эффективное управление памятью «из коробки».

Кроме того, в пятой версии `Spring Core` появился дополнительный класс `DataBufferUtils` для поддержки ввода/вывода (с сетью, ресурсами, файлами и т.д.) в форме реактивных потоков. Например, вот как можно реактивно прочитать трагедию В. Шекспира «Гамлет» с поддержкой обратного давления:

```
Flux<DataBuffer> reactiveHamlet = DataBufferUtils
    .read(
        new DefaultResourceLoader().getResource("hamlet.txt"),
        new DefaultDataBufferFactory(),
        1024
    );
```

Как видим, `DataBufferUtils.read` возвращает поток `Flux` экземпляров `DataBuffer`. Таким образом, мы можем использовать все возможности `Reactor` для чтения трагедии «Гамлет».

Наконец, последняя важная особенность, имеющая отношение к реактивности в `Spring Core`, – это **реактивные кодеки**. Они дают удобную возможность преобразования потока экземпляров `DataBuffer` в поток объектов и обратно. Для этой цели определены интерфейсы `Encoder` и `Decoder`, предлагающие следующий API для кодирования/декодирования потоков данных.

```
interface Encoder<T> {
    ...
    Flux<DataBuffer> encode(Publisher<? extends T> inputStream, ...);
}

interface Decoder<T> {
    ...
    Flux<T> decode(Publisher<DataBuffer> inputStream, ...);
    Mono<T> decodeToMono(Publisher<DataBuffer> inputStream, ...);
}
```

Как видим, оба интерфейса оперируют экземплярами `Publisher` и позволяют кодировать/декодировать поток экземпляров `DataBuffer` в объекты. Основное преимущество такого API заключается в возможности неблокирующего преобразо-

вания сериализованных данных в Java-объекты и обратно. Кроме того, такой способ кодирования/декодирования может уменьшить задержку обработки, потому что природа реактивных потоков позволяет обрабатывать элементы независимо, так что нет необходимости ждать получения последнего байта, чтобы начать декодирование всего набора данных. И наоборот, не обязательно иметь весь список объектов, чтобы начать их кодирование и отправку в канал ввода/вывода.



Узнать больше о поддержке реактивного ввода/вывода в Spring Core можно по ссылке <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#databuffers>.

Подводя итог, отметим, что пятая версия Spring Core в Spring Framework предлагает превосходный фундамент для реактивного программирования. Spring Boot, в свою очередь, предлагает этот фундамент в качестве базового компонента для любых приложений. Он позволяет писать реактивные приложения и прилагать меньше усилий для изобретения способов преобразования реактивных типов, организации ввода/вывода в реактивном стиле и кодирования/декодирования данных на лету.

Реактивность в Web

Также важно отметить существенные изменения в модуле Web. Прежде всего Spring Boot 2 предлагает новый модуль WebFlux, открывающий новые возможности для приложений с высокой пропускной способностью и низкой задержкой. Модуль Spring WebFlux построен на основе адаптера реактивных потоков и обеспечивает интеграцию с такими серверными движками, как Netty и Undertow, помимо поддержки обычных серверов на основе Servlet API 3.1. Проще говоря, Spring WebFlux предлагает неблокирующую основу и открывает новые возможности для Reactive Streams как центральной абстракции взаимодействий между бизнес-логикой и серверным движком.



Обратите внимание, что адаптер для Servlet API 3.1 обеспечивает полностью асинхронную и неблокирующую интеграцию, отличающуюся от адаптера WebMVC. Кроме того, модуль Spring WebMVC поддерживает Servlet API 4.0 с поддержкой HTTP/2.

Также Spring WebFlux широко использует Reactor 3. Благодаря этому мы получаем превосходную поддержку реактивного программирования «из коробки», не прилагая никаких дополнительных усилий, и можем запускать веб-приложения поверх встроенной интеграции Project Reactor с Netty. Наконец, модуль WebFlux предлагает встроенную поддержку обратного давления, благодаря которой можем сбалансировать ввод/вывод. Помимо изменений во взаимодействиях на стороне сервера Spring WebFlux добавляет новый класс WebClient, реализующий неблокирующие взаимодействия на стороне клиента.

Кроме того, старый добрый модуль WebMVC тоже получил некоторую поддержку реактивных потоков. Начиная с пятой версии фреймворка основой для модуля WebMVC стал Servlet API 3.1. Это означает, что теперь WebMVC поддерживает неблокирующий ввод/вывод в форме, предлагаемой стандартом Servlet. Однако дизайн модуля WebMVC мало изменился и не обеспечивает поддержку неблокирующего ввода/вывода на должном уровне. Тем не менее асинхронное поведение Servlet 3.0 было реализовано правильно. Для восполнения нехватки в поддержке реактивности Spring WebMVC предлагает обновленный класс `ResponseBodyEmitterReturnValueHandler`. Поскольку класс `Publisher` можно считать неисчерпаемым источником событий, обработчик `Emitter` лучше всего подходит для размещения логики обработки реактивных типов без нарушения всей инфраструктуры модуля WebMVC. Для этого в модуль WebMVC был добавлен класс `ReactiveTypeHandler`, который заботится о правильной обработке реактивных типов, таких как `Flux` и `Mono`.

Помимо поддержки реактивных типов на стороне сервера для получения неблокирующего поведения на стороне клиента можно использовать тот же класс `WebClient` из модуля WebFlux. На первый взгляд может показаться, что это вызовет конфликт между двумя модулями. К счастью, Spring Boot приходит на выручку в этой сложной ситуации и обеспечивает необходимое управление окружением, основанное на классах, доступных в `classpath`. Соответственно, подключая модуль WebMVC (`spring-boot-starter-web`) вместе с WebFlux, мы получаем окружение WebMVC и неблокирующую реактивность класса `WebClient` из модуля WebFlux.

Наконец, сравнив модули как реактивные конвейеры, мы получаем структуру, изображенную на рис. 5.1.

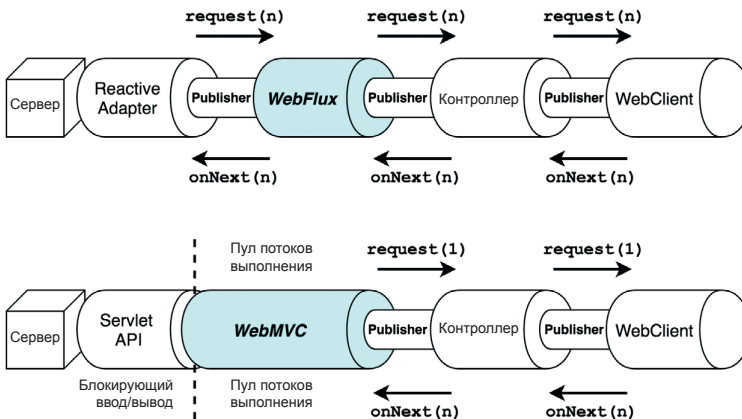


Рис. 5.1. Схематическое представление реактивного модуля WebFlux и частично реактивного модуля WebMVC в форме конвейеров

Как показано на рис. 5.1, в обоих случаях, используя WebMVC или WebFlux, мы получаем почти идентичную модель программирования, основанную на стан-

дарте Reactive Streams. Одно из самых заметных отличий между модулями заключается в том, что WebMVC требует блокирования операций чтения/записи в точке интеграции с Servlet API, что обусловлено историческими предпосылками. Это ухудшает модель взаимодействий в реактивном потоке и упрощает ее до модели *PULL*. Кроме того, модуль WebMVC теперь использует пул потоков выполнения для всех блокирующих операций ввода/вывода. Как следствие его необходимо правильно настроить, чтобы избежать непредвиденного поведения.

Модель взаимодействий в WebFlux, напротив, полагается на пропускную способность Сети и базовые транспортные протоколы, которые могут определять свой поток управления.

Подводя итог, скажем, что Spring 5 представляет собой мощный набор инструментов для создания реактивных и неблокирующих приложений с использованием Project Reactor. Кроме того, Spring Boot обеспечивает надежное управление зависимостями и автоматические настройки, защищая нас от ада зависимостей. Мы не будем подробно обсуждать новые реактивные возможности, появившиеся в модуле Web, но подробно исследуем модуль WebFlux в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux».

Реактивность в Spring Data

Изменения были внесены не только в веб-слой. Не менее важные изменения для большинства приложений внесли в слой данных, который реализует взаимодействия с хранилищами. В течение многих лет надежным решением, упрощающим разработку приложений, оставался проект Spring Data, предлагающий удобные абстракции для доступа к данным. С самого начала Spring Data обеспечивал по большей части синхронный, блокирующий доступ к хранилищам. К счастью, пятое поколение фреймворка Spring Data получило новые возможности, обеспечивающие реактивный и неблокирующий доступ к слою базы данных. Новая версия Spring Data предлагает интерфейс `ReactiveCrudRepository` с бесшовной интеграцией с реактивными типами Project Reactor. Как следствие это превратило соединения с базами данных в эффективные компоненты реактивных приложений.

Помимо реактивного репозитория Spring Data предлагает несколько модулей, интегрированных с методами хранения посредством интерфейса `ReactiveCrudRepository`. Ниже приводится список хранилищ, для которых теперь в Spring Data имеется реактивная интеграция:

- **MongoDB через модуль Spring Data Mongo Reactive.** Обеспечивает полностью реактивные и неблокирующие взаимодействия с базой данных NoSQL, а также надежное управление обратным давлением;
- **Cassandra через модуль Spring Data Cassandra Reactive.** Обеспечивает реактивные и неблокирующие взаимодействия с хранилищем данных

Cassandra, а также поддерживает управление обратным давлением по протоколу TCP;

- **Redis через модуль Spring Data Redis Reactive** – реактивная интеграция с Redis посредством Java-клиента Lettuce;
- **Couchbase через модуль Spring Data Couchbase Reactive** – реактивная интеграция Spring Data с базой данных Couchbase через драйвер на основе RxJava.

Кроме того, все эти модули поддерживаются фреймворком Spring Boot, в который входят дополнительные модули с настройками, обеспечивающие гладкую интеграцию с выбранным хранилищем.

Наряду с поддержкой баз данных NoSQL Spring Data предлагает Spring Data JDBC – облегченную интеграцию с JDBC, в которой вскоре может появиться поддержка реактивности. Более подробно о реактивном доступе к данным мы поговорим в главе 7 «Реактивный доступ к базам данных».

Подводя итог, отметим, что пятое поколение Spring Data гарантирует сквозную реактивность потоков данных от конечных точек до баз данных и покрывает потребности большинства приложений. Кроме того, как будет показано в следующих разделах, большинство улучшений в других модулях Spring Framework описывается на реактивные возможности модуля WebFlux или модуля Spring Data.

Реактивность в Spring Session

Еще одно важное обновление в Spring Framework, связанное с модулем Spring Web, – добавление поддержки реактивности в модуль Spring Session.

Теперь мы имеем поддержку модуля WebFlux и можем использовать эффективную абстракцию для управления сеансами. С этой целью в Spring Session был добавлен класс `ReactiveSessionRepository`, обеспечивающий асинхронный и неблокирующий доступ к хранимым сеансам с помощью типа `Mono` из библиотеки `Reactor`.

Кроме того, Spring Session предлагает реактивную интеграцию с Redis в качестве хранилища, взаимодействие с которым осуществляется с использованием реактивных инструментов Spring Data. Благодаря этому можно получить распределенные `WebSession`, просто включив следующие зависимости.

```
compile "org.springframework.session:spring-session-data-redis"
compile "org.springframework.boot:spring-boot-starter-webflux"
compile "org.springframework.boot:spring-boot-starter-data-redis-reactive"
```

Как можно судить по предыдущим зависимостям, чтобы организовать реактивное управление `WebSession`, нужно объединить эти три зависимости. Кроме того,

Spring Boot обеспечит создание точной комбинации компонентов и автоматическую настройку для бесперебойной работы веб-приложения.

Реактивность в Spring Security

Для поддержки модуля WebFlux фреймворк Spring 5 предлагает улучшенную поддержку реактивности в модуле Spring Security. Главным улучшением в этом модуле стала поддержка модели реактивного программирования, продвигаемой проектом Project Reactor. Если помните, прежняя версия Spring Security использовала `ThreadLocal` для хранения экземпляров `SecurityContext`. Этот прием хорошо работал, когда выполнение протекало в одном потоке `Thread`. В любой момент мы могли получить доступ к `SecurityContext`, хранящемуся внутри `ThreadLocal`. Однако с внедрением методов асинхронных взаимодействий возникла проблема – мы должны приложить дополнительные усилия, чтобы обеспечить передачу содержимого `ThreadLocal` в другой поток выполнения, и делать это для каждого экземпляра при каждом переключении между потоками выполнения. Несмотря на то что Spring Framework упрощает передачу `SecurityContext` между потоками выполнения с использованием дополнительного расширения `ThreadLocal`, у нас все еще могут возникать проблемы при использовании парадигмы реактивного программирования с Project Reactor или похожими на нее библиотеками.

К счастью, новое поколение Spring Security применяет новый механизм из библиотеки Reactor для передачи контекста безопасности в потоках данных Flux и Mono. Благодаря этому мы можем свободно обращаться к данному контексту даже в очень сложных реактивных потоках данных, которые могут действовать в разных потоках выполнения. Более подробно о реализации этой возможности в реактивном стеке рассказывается в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux».

Реактивность в Spring Cloud

Экосистема Spring Cloud нацелена на создание реактивных систем с использованием Spring Framework, однако парадигма реактивного программирования не обошла стороной и Spring Cloud. Прежде всего изменения коснулись точки входа в распределенную систему, называемую **шлюзами**. Долгое время единственным модулем в Spring, который позволял запускать приложение в роли шлюза, был модуль **Spring Cloud Netflix Zuul**. Как вы, наверное, знаете, Netflix Zuul основан на Servlet API, использующем блокирующую и синхронную маршрутизацию запросов. Единственный способ распараллелить обработку запросов и улучшить производительность – настроить пул потоков выполнения на сервере. К сожалению, такая модель масштабируется хуже, чем реактивный подход, а описание причин вы найдете в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux».

К счастью, в Spring Cloud появился новый модуль Spring Cloud Gateway, реализованный на основе Spring WebFlux и предлагающий асинхронную и неблокирующую маршрутизацию с использованием поддержки Project Reactor 3.



Узнать больше о Spring Cloud Gateway можно по ссылке <https://cloud.spring.io/spring-cloud-gateway>.

Кроме нового модуля Gateway в Spring Cloud Streams появилась поддержка Project Reactor и более детальная модель управления потоками данных. Подробнее рассмотрим Spring Cloud Streams в главе 8 «*Масштабирование с Cloud Streams*».

Наконец, чтобы упростить разработку реактивных систем, в Spring Cloud был добавлен новый модуль с именем Spring Cloud Function, предлагающий основные компоненты, необходимые для реализации своих решений, – **функция как служба** (Function as a Service, FaaS). Как будет показано в главе 8 «*Масштабирование с Cloud Streams*», для использования модуля Spring Cloud Function требуется подготовить дополнительную инфраструктуру. К счастью, Spring Cloud Data Flow предлагает все необходимое и включает часть возможностей Spring Cloud Function. Здесь мы не будем вдаваться в детали Spring Cloud Function и Spring Cloud Data Flow, потому что более подробно они рассматриваются в главе 8 «*Масштабирование с Cloud Streams*».

Реактивность в Spring Test

Неотъемлемой частью любого процесса разработки является тестирование, поэтому экосистема Spring предлагает усовершенствованные модули, Spring Test и Spring Boot Test, с обширным списком дополнительных возможностей для тестирования реактивных Spring-приложений. Модуль Spring Test предлагает класс `WebTestClient` для тестирования веб-приложений на основе WebFlux, а Spring Boot Test – средства автоматической настройки наборов тестов в виде обычных аннотаций.

Кроме того, для тестирования реализаций `Publisher` библиотека Project Reactor предлагает модуль **Reactor-Test**, который в комбинации с модулями Spring Test и Spring Boot Test позволяет писать законченные наборы тестов для проверки бизнес-логики, реализованной с использованием реактивных инструментов из Spring. Более подробно о тестировании реактивных приложений поговорим в главе 9 «*Тестирование реактивных приложений*».

Реактивность в мониторинге

Наконец, всякая реактивная система промышленного уровня, построенная с использованием Project Reactor и реактивных инструментов Spring Framework, должна обеспечивать доступ ко всем метрикам, характеризующим ее функционирование. Для этого в экосистеме Spring имеется несколько механизмов мониторинга приложений с разной степенью детализации.

Прежде всего в самой библиотеке Project Reactor есть встроенные средства поддержки мониторинга. Она предлагает метод `Flux#metrics()` для трассировки разных событий в реактивном потоке данных. Однако, помимо точек мониторинга, регистрируемых вручную, типичное веб-приложение должно отслеживать работу множества внутренних процессов и некоторым способом сообщать свои оперативные показатели. С этой целью экосистема Spring Framework предлагает обновленный модуль Spring Boot Actuator, который помогает получить основные показатели, полезные для мониторинга приложения и устранения проблем. Новое поколение Spring Actuator обеспечивает полную интеграцию с WebFlux и использует асинхронную и неблокирующую модель программирования для эффективного получения метрик.

Последний инструмент для мониторинга и трассировки приложения, работающий «из коробки», предлагается модулем **Spring Cloud Sleuth**. Заметным улучшением здесь является поддержка реактивного программирования с использованием Project Reactor, благодаря чему правильно отслеживаются все реактивные процессы.

Подводя итог, можно сказать, что реактивные усовершенствования в ядре фреймворка и во всей экосистеме Spring обеспечивают средства и возможности полноценного мониторинга реактивных решений. Все эти аспекты мы более подробно рассмотрим в главе 10 «И наконец, выпуск!».

Заключение

Как было показано в этой главе, библиотека Spring Boot создавалась с целью упростить разработку приложений на основе Spring Framework. Она действует в роли связующего звена между компонентами Spring и предлагает работоспособные конфигурации по умолчанию, исходя из зависимостей приложения. Версия 2 библиотеки обеспечивает также превосходную поддержку реактивного стека. Здесь мы не стали углубляться в детали, касающиеся улучшений Spring Framework, и рассказали о том, как Spring Boot помогает с легкостью получить все преимущества реактивного программирования.

Однако в следующих главах мы подробно исследуем новые возможности и улучшения в Spring 5.x, начав с модуля Spring WebFlux и сравнив его со старым модулем Spring WebMVC.

Глава 6

Неблокирующие и асинхронные взаимодействия с WebFlux

В предыдущей главе мы познакомились с библиотекой Spring Boot 2.x. Мы увидели, что в пятой версии Spring Framework появилось множество полезных обновлений, а также познакомились с модулем Spring WebFlux.

В этой главе подробно исследуем данный модуль. Сравним внутренний дизайн WebFlux с дизайном старого доброго модуля Web MVC и попробуем оценить сильные и слабые стороны обоих. Также создадим здесь простое приложение на основе WebFlux.

Будут рассмотрены следующие темы:

- общий обзор Spring WebFlux;
- сравнение Spring WebFlux и Spring Web MVC;
- детальный обзор дизайна модуля Spring WebFlux.

WebFlux как основа реактивного сервера

Как мы видели в главе 1 «*Причины выбора Spring*» и в главе 4 «*Project Reactor – основа реактивных приложений*», новая эра серверов приложений принесла новые приемы разработки. В самом начале освоения Spring Framework сферы разработки веб-приложений было принято решение интегрировать модуль Spring Web с Java EE Servlet API. Вся инфраструктура Spring Framework построена на механизме Servlet API и тесно связана с ним. Например, модуль Spring Web MVC целиком основан на шаблоне «*Единая точка входа (Front Controller)*». В Spring Web MVC этот шаблон реализует класс `org.springframework.web.servlet.DispatcherServlet`, который косвенно наследует класс `javax.servlet.http.HttpServlet`.

С другой стороны, Spring Framework дает нам лучший уровень абстракции в модуле *Spring Web*, который служит строительным блоком для многих механизмов, таких как контроллеры, управляемые аннотациями. Даже притом что этот модуль частично отделяет общие интерфейсы от их реализаций, первоначальный дизайн Spring Web тоже был основан на модели синхронных взаимодействий и, как следствие, блокирующем вводе/выводе. Тем не менее такое разделение является хорошей основой, поэтому, прежде чем перейти к исследованию реактивных возможностей, рассмотрим в общих чертах дизайн модуля WebMVC и попытаемся понять, как он работает (см. рис. 6.1).

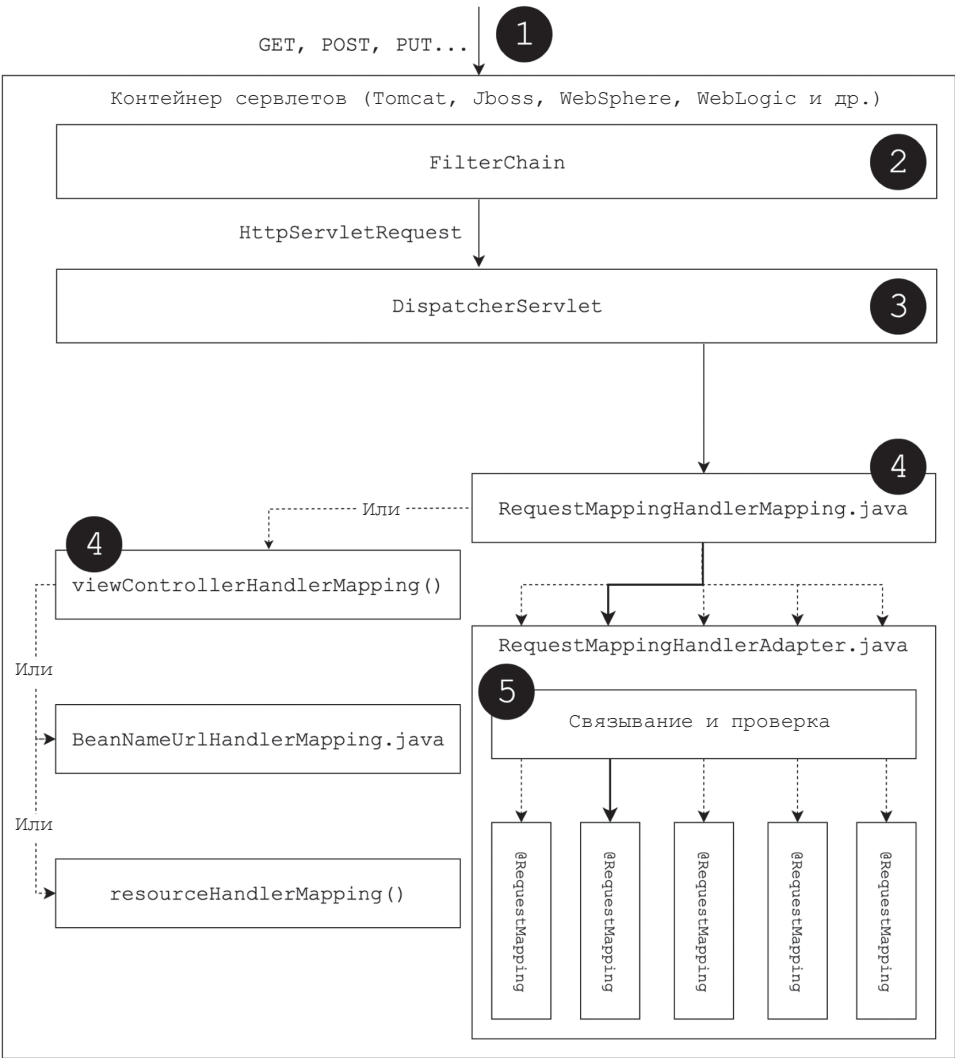


Рис. 6.1. Реализация веб-стека в модуле Spring WebMVC

Остановимся на некоторых важных моментах, изображенных на этой диаграмме.

1. Входящий запрос обрабатывается базовым **контейнером сервлетов**. В данном случае контейнер сервлетов отвечает за преобразование тела входящего запроса в экземпляр интерфейса `ServletRequest` и ответа в экземпляр интерфейса `ServletResponse`.
2. Этап фильтрации `ServletRequest` с использованием фильтров, объединенных в `FilterChain`.
3. Следующий этап – обработка `DispatcherServlet`. Напомним, что `DispatcherServlet` наследует класс `Servlet`. Он также хранит списки `HandlerMappings` (4), `HandlerAdapters` (5) и `ViewResolvers` (не обозначен на схеме). В контексте текущего потока выполнения класс `DispatcherServlet` отвечает за поиск экземпляра `HandlerMapping` и его преобразование с использованием подходящего экземпляра `HandlerAdapter`. Затем он ищет `ViewResolver`, который, в свою очередь, найдет представление `View`, чтобы `DispatcherServlet` мог инициировать отображение результатов выполнения `HandlerMapping` и `HandlerAdapter`.
4. Этап `HandlerMapping`. `DispatcherServlet` (3) отыскивает все компоненты `HandlerMapping` в контексте приложения. Все найденные экземпляры сортируются по порядковым номерам. Порядковые номера определяются аннотацией `@Order` или реализацией интерфейса `Ordered` в `HandlerMapping`. Вследствие этого поиск подходящих экземпляров `HandlerMapping` зависит от порядка, который был задан заранее. На рис. 6.1 изображено несколько наиболее широко используемых экземпляров `HandlerMapping`. Самый известный из них – `RequestMappingHandlerMapping`, он обеспечивает модель программирования на основе аннотаций.
5. Последний этап – `RequestMappingHandlerAdapter`, который гарантирует правильную привязку входящего `ServletRequest` к объекту с аннотацией `@Controller`. Вдобавок к этому `RequestMappingHandlerAdapter` обеспечивает проверку запроса, преобразование ответа и множество других полезных действий, которые делают фреймворк Spring Web MVC таким привлекательным для веб-разработчиков.

Как нетрудно заметить, вся конструкция опирается на базовый контейнер сервлетов, отвечающий за обработку всех отображенных сервлетов внутри контейнера. `DispatchServlet` действует как точка интеграции между гибкой и легко настраиваемой инфраструктурой Spring Web и сложным и тяжелым `Servlet API`. Настраиваемая абстракция `HandlerMapping` помогает отделить конечную бизнес-логику, например контроллеры и компоненты, от `Servlet API`.



TIP

Spring MVC поддерживает прямое взаимодействие с `HttpServletRequest` и `HttpServletResponse` наряду с возможностями отображения, привязки и проверки. Однако при использовании этих классов возникает дополнительная прямая зависимость от `Servlet API`. Это можно считать плохой практикой, потому что такая зависимость может усложнить процесс пе-

перехода с Web MVC на WebFlux или любое другое расширение для Spring. По этой причине рекомендуется использовать классы `org.springframework.http.ResponseEntity` и `org.springframework.http.ResponseEntity`. Они изолируют объекты запросов и ответов от реализации веб-сервера.

Модель программирования с использованием Spring Web MVC долгие годы считалась одной из самых удобных. Этот фреймворк доказал свою надежность и стабильность в веб-разработке. Вот почему в 2003 году Spring Framework встал на путь, который сделал его одним из самых популярных решений для создания веб-приложений поверх Servlet API. Однако приемы и методы прошлого не соответствуют требованиям современных систем, интенсивно использующих данные.

Несмотря на то что Servlet API поддерживает асинхронные, неблокирующие взаимодействия (начиная с версии 3.1), реализация модуля Spring MVC имеет множество пробелов и не поддерживает неблокирующие операции в жизненном цикле запроса. Например, отсутствует готовый неблокирующий HTTP-клиент, поэтому любая попытка взаимодействия извне наверняка приведет к выполнению блокирующей операции ввода/вывода. Как отмечалось в главе 5 «Добавление реактивности с помощью *Spring Boot 2*», абстракция Web MVC поддерживает не все возможности неблокирующего Servlet API 3.1. Пока этого не случится, фреймворк Spring Web MVC не может рассматриваться как основа для высоконагруженных приложений. Еще один недостаток старой абстракции Spring – отсутствие гибкости в повторном использовании функций Spring Web или моделей программирования для серверов, не связанных с сервлетами, таких как Netty.

Вот почему в последние годы главной задачей разработчиков Spring Framework было создание нового решения, поддерживающего ту же модель программирования с применением аннотаций и в то же время предлагающую все преимущества использования неблокирующих серверов.

Реактивное веб-ядро

Представьте, что мы работаем над созданием нового асинхронного и неблокирующего веб-модуля для новой экосистемы Spring. Как должен выглядеть новый реактивный веб-стек? Для начала проанализируем существующие решения и определим части, которые следует улучшить или убрать.

Следует отметить, что в целом внутренний API фреймворка Spring MVC имеет хороший дизайн. Единственное, что нужно добавить в него, – это прямая зависимость от Servlet API. То есть окончательное решение должно включать интерфейсы, напоминающие интерфейсы из Servlet API. Первый шаг к созданию реактивного стека – замена `javax.servlet.Servlet#service` аналогичным интерфейсом с методом, обрабатывающим входящие запросы. Также необходимо заменить родственные интерфейсы и классы и улучшить способ, каким Servlet API заменяет запрос клиента ответом сервера.

Введение собственного API позволит нам разорвать тесную связь с серверными движками и конкретными API, но никак не поможет организовать реактивные взаимодействия. Следовательно, все новые интерфейсы должны обеспечивать доступ ко всем данным, таким как тело запроса и сеанс, в реактивном формате. Как мы узнали в предыдущих главах, модель реактивных потоков позволяет взаимодействовать и обрабатывать данные с учетом доступности и потребности в них. Библиотека Project Reactor следует положениям стандарта Reactive Streams и предлагает обширный API с точки зрения возможностей, поэтому она может оказаться подходящим инструментом для создания всех реактивных веб-API.

Наконец, объединив эти идеи в реальной реализации, мы получим следующий код:

```
interface ServerHttpRequest {                                // (1)
    ...                                                      //
    Flux<DataBuffer> getBody();                               // (1.1)
    ...                                                      //
}                                                            //

interface ServerHttpResponse {                               // (2)
    ...                                                      //
    Mono<Void> writeWith(Publisher<? extends DataBuffer> body); // (2.1)
    ...                                                      //
}                                                            //

interface ServerWebExchange {                                // (3)
    ...                                                      //
    ServerHttpRequest getRequest();                          // (3.1)
    ServerHttpResponse getResponse();                        // (3.2)
    ...                                                      //
    Mono<WebSession> getSession();                            // (3.3)
    ...                                                      //
}                                                            //
```

Пояснения к коду.

1. Макет интерфейса, представляющего входящее сообщение. Как можно видеть в строке (1.1), роль главной абстракции, открывающей доступ к входящим байтам, играет Flux, по определению имеющий реактивный доступ. Как рассказывалось в главе 5 «Добавление реактивности с помощью Spring Boot 2», DataBuffer – отличная абстракция буфера байтов. Она предлагает удобную возможность обмена данными с конкретной реализацией сервера. Кроме тела любой HTTP-запрос содержит также информацию о заголовках, путях, cookies и параметрах запроса, поэтому доступ к данной информации можно организовать посредством отдельных методов в этом интерфейсе или в подчиненных интерфейсах.
2. Макет интерфейса ответа, сопутствующего интерфейсу ServerHttpRequest. Как можно видеть в строке (2.1), в отличие от метода ServerHttpRequest

`Request#getBody`, метод `ServerHttpResponse#writeWith` принимает любой класс `Publisher<? extends DataBuffer>`. В данном случае реактивный тип `Publisher` дает нам больше гибкости и избавляет от тесной зависимости от этой конкретной реактивной библиотеки, благодаря чему мы можем использовать любую реализацию интерфейса и отделить бизнес-логику от фреймворка. Метод возвращает экземпляр `Mono<Void>`, представляющий асинхронный процесс отправки данных в сеть. Здесь важно отметить, что процесс отправки данных выполняется, только когда мы подпишемся на данный поток `Mono`. Кроме того, принимающий сервер может управлять обратным давлением, используя поток управления транспортным протоколом.

3. Объявление интерфейса `ServerWebExchange`. Здесь интерфейс действует как контейнер для экземпляров запросов и ответов HTTP (3.1 и 3.2). Интерфейс является инфраструктурным и, так же как взаимодействие HTTP, может хранить информацию, имеющую отношение к фреймворку. Например, он может хранить информацию о веб-сеансе `WebSession`, восстановленном из входящего запроса, как показано в строке (3.3). С другой стороны, он может определять дополнительные инфраструктурные методы поверх интерфейсов запроса и ответа.

В предыдущем примере мы набросали потенциальные интерфейсы для нашего реактивного веб-стека. Вообще, эти три интерфейса похожи на имеющиеся в `Servlet API`. Например, `ServerHttpRequest` и `ServerHttpResponse` могут показаться похожими на `ServletRequest` и `ServletResponse`. По сути, реактивные аналоги нацелены на объявление практически тех же методов с точки зрения модели взаимодействий. Однако из-за асинхронной и неблокирующей природы реактивных потоков мы получаем готовую основу с потоковой передачей информации и защиту от сложных API на основе обратных вызовов. Этот подход также защищает нас от ада обратных вызовов.

Помимо центральных интерфейсов для организации всего спектра взаимодействий мы должны определить интерфейсы обработчиков запросов/ответов и фильтров.

```
interface WebHandler {                                     // (1)
    Mono<Void> handle(ServerWebExchange exchange);         //
}                                                         //

interface WebFilterChain {                                 // (2)
    Mono<Void> filter(ServerWebExchange exchange);         //
}                                                         //

interface WebFilter {                                     // (3)
    Mono<Void> filter(ServerWebExchange exch, WebFilterChain chain); //
}                                                         //
```

Вот некоторые пояснения к этому коду.

1. Центральная точка входа в любые HTTP-взаимодействия называется *Web Handler*. Наш интерфейс играет роль абстрактного класса *Dispatcher Servlet*, что позволяет нам определить на его основе любую реализацию. Поскольку интерфейс отвечает за поиск обработчика запроса и его последующую компоновку с представлением, которое выведет результаты выполнения в *ServerHttpResponse*, метод *DispatchServlet#handle* не обязан возвращать какие-либо результаты. Однако иногда бывает полезно иметь сигнал о завершении обработки. Опираясь на такой сигнал, можно установить тайм-ауты для обработки и прекращать выполнение, если сигнал не поступил в течение некоторого времени. По этой причине метод возвращает *Mono* из *Void*, что позволяет получить уведомление о завершении асинхронной обработки без необходимости обрабатывать результат.
2. Этот интерфейс позволяет объединить в цепочку несколько экземпляров *WebFilter* (3), так же как в *Servlet API*.
3. Реактивное представление класса *Filter*.

Предыдущие объявления образуют основу, на которой мы можем начать строить бизнес-логику остальной части фреймворка.

Мы определили почти все основные элементы реактивной веб-инфраструктуры. Чтобы завершить иерархию абстракций, нужно добавить контракт самого низкого уровня для реактивной обработки HTTP-запросов. Поскольку выше мы объявили только интерфейсы, ответственные за передачу и обработку данных, нам следует определить интерфейс, отвечающий за адаптацию серверного движка под нашу инфраструктуру. Для этого понадобится дополнительный уровень абстракции, который будет отвечать за прямые взаимодействия с *ServerHttpRequest* и *ServerHttpResponse*.

Кроме того, данный уровень должен отвечать за конструирование *ServerWebExchange*. Здесь же находится конкретное хранилище сеансов, региональные настройки и другие подобные инфраструктурные элементы.

```
public interface HttpHandler {  
    Mono<Void> handle(  
        ServerHttpRequest request,  
        ServerHttpResponse response);  
}
```

Наконец, для каждого серверного движка мы можем иметь адаптер, вызывающий промежуточный *HttpHandler*, который затем создает требуемые экземпляры *ServerHttpResponse* и *ServerHttpRequest* для *ServerWebExchange* и передает их в *WebFilterChain* и *WebHandler*. С таким дизайном для пользователей *Spring WebFlux* становится не важно, как работает конкретный серверный движок, потому что теперь у нас имеется надлежащий уровень абстракции, скрывающий тех-

нические детали серверного движка. Мы можем сделать следующий шаг и определить реактивные абстракции верхнего уровня.

Реактивные фреймворки Web и MVC

Как уже отмечалось, ключевой особенностью модуля Spring Web MVC является модель программирования на основе аннотаций, поэтому главной задачей является реализация той же идеи и в реактивном веб-стеке. Если посмотреть на текущий модуль Spring Web MVC, можно увидеть, что в целом он спроектирован правильно. Поэтому вместо определения новой реактивной инфраструктуры MVC можно повторно использовать имеющуюся инфраструктуру, заменив синхронные взаимодействия реактивными типами, такими как Flux, Mono и Publisher. Например, двумя центральными интерфейсами для привязки контекстной информации (заголовков, параметров запроса, атрибутов и сеансов) и отображения запросов в найденные обработчики являются HandlerMapping и HandlerAdapter. В общем и целом мы можем сохранить ту же цепочку из HandlerMapping и HandlerAdapter, как в Spring Web MVC, но заменить синхронный императив реактивными взаимодействиями с использованием типов из библиотеки Reactor.

```
interface HandlerMapping {                                     // (1)
    /* HandlerExecutionChain getHandler(HttpServletRequest request) */ // (1.1)
    Mono<Object> getHandler(ServerWebExchange exchange);        // (1.2)
}                                                             //

interface HandlerAdapter {                                    // (2)
    boolean supports(Object handler);                          //
                                                            //
    /* ModelAndView handle(                                    // (2.1)
        HttpServletRequest request, HttpServletResponse response, //
        Object handler                                           //
    ) */                                                         //

    Mono<HandlerResult> handle(                                   // (2.2)
        ServerWebExchange exchange,                             //
        Object handler                                           //
    );                                                         //
}                                                             //
```

Пояснения к коду.

1. Объявление реактивного интерфейса HandlerMapping. Здесь, чтобы подчеркнуть отличия между старой реализацией Web MVC и улучшенной, приводятся оба объявления методов. Старая реализация (1.1) закомментирована с использованием `/* ... */` и выделена *курсивом*, новый интерфейс (1.2) выделен **жирным** шрифтом. Как видите, в целом методы очень похожи

и отличаются только тем, что последний возвращает тип `Mono`, что обеспечивает поддержку реактивного поведения.

2. Реактивная версия интерфейса `HandlerAdapter`. Как можно заметить, реактивная версия метода `handle` немного лаконичнее, потому что класс `ServerWebExchange` объединяет экземпляры запроса и ответа. В строке (2.2) метод возвращает `Mono` из `HandlerResult` вместо `ModelAndView` (2.1). Как мы помним, `ModelAndView` отвечает за поддержку такой информации, как код состояния, `Model` и `View`. Класс `HandlerResult` содержит ту же информацию, кроме кода состояния. Но `HandlerResult` лучше, потому что предлагает результат прямого выполнения, поэтому экземпляру `DispatcherHandler` проще найти требуемый обработчик. В Web MVC интерфейс `View` отвечает за шаблоны и за объекты, а также отображает результаты, поэтому его назначение в Web MVC может показаться немного туманным. К сожалению, такие многочисленные обязанности непросто адаптировать к асинхронной обработке результатов. В подобных случаях, когда результат является обычным Java-объектом, поиск `View` выполняется в `HandlerAdapter`, что не является прямой ответственностью этого класса. Следовательно, лучше постараться сохранить ответственность как можно более отчетливой, а значит, изменение, реализованное в предыдущем коде, действительно является улучшением.

Следуя этим путем, мы получим модель реактивных взаимодействий, не нарушив иерархии выполнения, сохранив существующий дизайн и повторно используя имеющийся код с минимальными изменениями.

Наконец, собрав воедино все, что мы сделали для создания реактивного веб-стека и корректировки потока обработки запроса с учетом реальной реализации, мы приходим к архитектуре, изображенной на рис. 6.2.

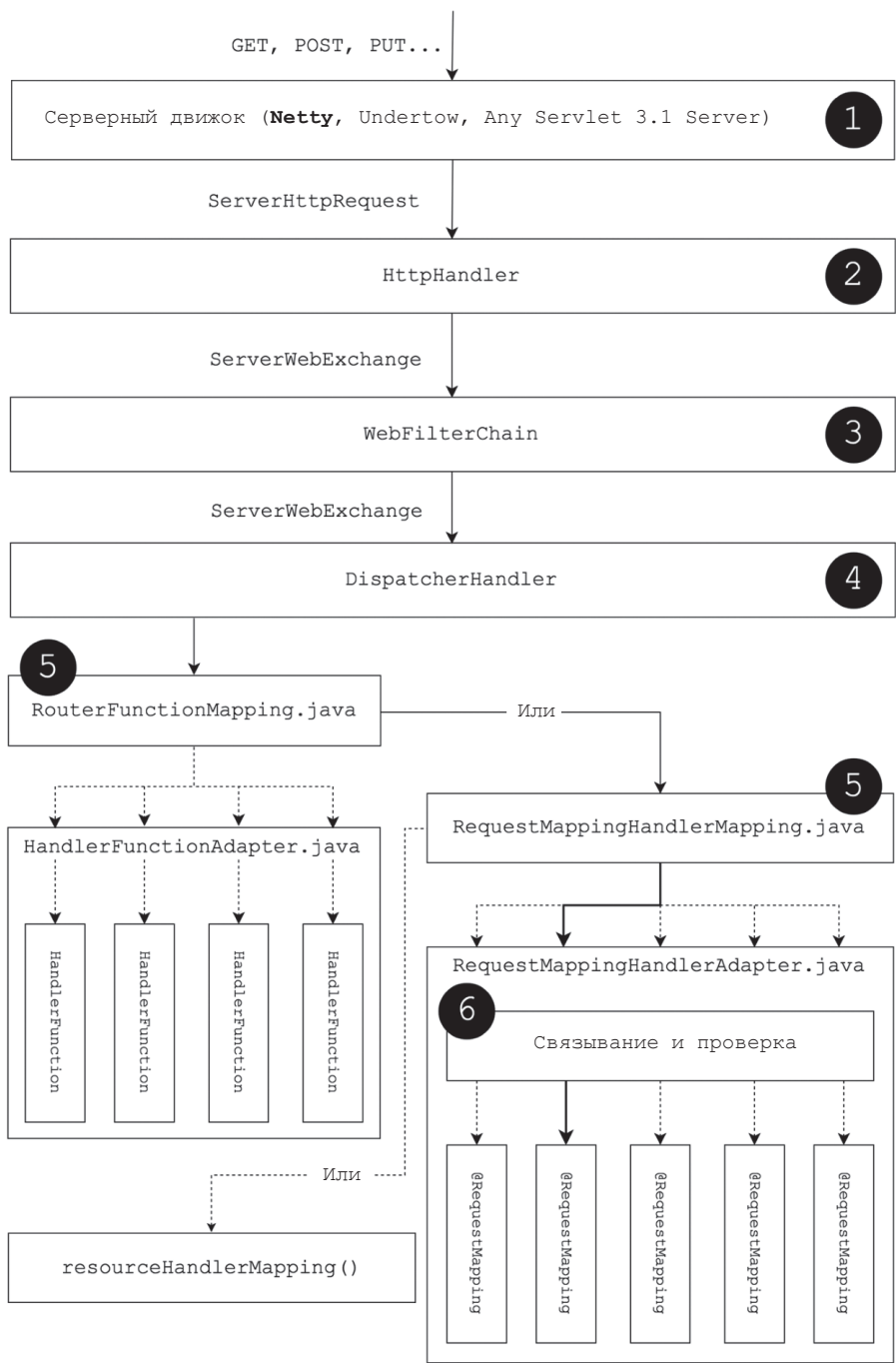


Рис. 6.2. Переработанный реактивный веб- и MVC-стек

Пояснения к диаграмме на рис. 6.2.

1. Входящий запрос, обрабатываемый **серверным движком**. Как видите, список серверных движков не ограничивается серверами с Servlet API и теперь включает такие движки, как *Netty* и *Undertow*. Здесь для каждого серверного движка имеется свой реактивный адаптер, который отображает внутреннее представление HTTP-запросов и HTTP-ответов в *ServerHttpRequest* и *ServerHttpResponse*.
2. Этап выполнения *HttpHandler*, который объединяет *ServerHttpRequest*, *ServerHttpResponse* пользовательский сеанс и прочую сопутствующую информацию в экземпляр *ServerWebExchange*.
3. Этап выполнения *WebFilterChain*, который добавляет в цепочку заданный *WebFilter*. Затем *WebFilterChain* вызывает метод *WebFilter#filter* каждого экземпляра *WebFilter* в этой цепочке, чтобы отфильтровать входящие *ServerWebExchange*.
4. Если все условия, определяемые фильтрами, соблюдены, *WebFilterChain* вызывает экземпляр *WebHandler*.
5. Следующий шаг – поиск экземпляров *HandlerMapping* и вызов первого подходящего. В этом примере мы изобразили несколько экземпляров *HandlerMapping*, таких как *RouterFunctionMapping*, *RequestMappingHandlerMapping* и *HandlerMapping*. Здесь присутствует пока незнакомый экземпляр *HandlerMapping* – *RouterFunctionMapping*, который появился в модуле *WebFlux* и выходит за рамки чисто функциональной обработки запросов. Поближе познакомимся с этим типом в следующем разделе.
6. Этап выполнения *RequestMappingHandlerAdapter*, который сохранил прежние функциональные обязанности, но теперь использует реактивные потоки для управления реактивными взаимодействиями.

Диаграмма на рис 6.2 дает лишь упрощенное представление о том, как протекают взаимодействия в модуле *WebFlux*. Следует отметить, что в модуле *WebFlux* по умолчанию предполагается работа с серверным движком *Netty*, потому что сервер *Netty* широко используется в реактивном пространстве. Кроме того, этот серверный движок обеспечивает асинхронные и неблокирующие взаимодействия между клиентом и сервером. То есть движок лучше всего вписывается в парадигму реактивного программирования, предлагаемую модулем *Spring WebFlux*. Но, даже притом что *Netty* является серверным движком, используемым по умолчанию, модуль *WebFlux* дает нам возможность выбрать другой серверный движок, а это значит, что мы легко можем переключаться между разными современными серверными движками, такими как *Undertow*, *Tomcat*, *Jetty*, или любыми движками на основе Servlet API. Как видите, модуль *WebFlux* во многом повторяет архитектуру модуля *Spring Web MVC*, поэтому его легко освоит любой человек, имеющий опыт использования прежнего веб-фреймворка. Кроме того, модуль *Spring Web Flux* содержит множество скрытых сокровищ, о которых пойдет речь в следующих разделах.

Чисто функциональные приемы в WebFlux

Как можно заметить на диаграмме, изображенной на рис. 6.2, несмотря на существенное сходство с Web MVC, модуль WebFlux предлагает массу новых возможностей. В эпоху повсеместного распространения микросервисов, Amazon Lambda и похожих облачных служб важно обеспечить поддержку функциональности, которая позволит разработчикам создавать легковесные приложения, использующие почти весь арсенал средств фреймворка. Одной из особенностей, добавившей привлекательности конкурирующим фреймворкам, таким как Vert.x и Ratpack, стала возможность создавать легковесные приложения благодаря функциональному стилю отображения маршрутов и встроенному API, с помощью которого можно определять сложную логику маршрутизации запросов. Именно по этой причине разработчики Spring Framework решили внедрить аналогичную возможность в модуль WebFlux. Кроме того, чисто функциональная маршрутизация прекрасно сочетается с новыми подходами реактивного программирования. Например, посмотрим, как определить сложную маршрутизацию, используя новый функциональный подход.

```
import static ...RouterFunctions.nest;           // (1)
import static ...RouterFunctions.nest;         //
import static ...RouterFunctions.route;        //
...
import static ...RequestPredicates.GET;        // (2)
import static ...RequestPredicates.POST;       //
import static ...RequestPredicates.accept;     //
import static ...RequestPredicates.contentType; //
import static ...RequestPredicates.method;     //
import static ...RequestPredicates.path;       //

@SpringBootApplication                           // (3)
public class DemoApplication {                  //
    ...
    @Bean
    public RouterFunction<ServerResponse> routes( // (4)
        OrderHandler handler                    // (4.1)
    ) {                                         //
        return
            nest(path("/orders"),              // (5)
                nest(accept(APPLICATION_JSON), //
                    route(GET("/{id}"), handler::get) //
                    .andRoute(method(HttpMethod.GET), handler::list) //
                )
                .andNest(contentType(APPLICATION_JSON), //
                    route(POST("/"), handler::create) //
                )
            )
    }
```

```

    });
}
}

```

Пояснения к коду.

1. Инструкции статического импорта из класса `RouterFunctions`. Как видите, класс `RouterFunctions` предлагает обширный список фабричных методов, которые возвращают интерфейсы `RouterFunction` с разным поведением.
2. Инструкции статического импорта из класса `RequestPredicates`. Как можно судить по предыдущему фрагменту кода, класс `RequestPredicates` позволяет проверять входящие запросы на соответствие самым разным условиям. В целом `RequestPredicates` обеспечивает доступ к разным реализациям интерфейса `RequestPredicate`, который является функциональным интерфейсом и может легко расширяться для реализации дополнительных нестандартных проверок.
3. Типичное объявление приложения Spring Boot, класс которого снабжен аннотацией `@SpringBootApplication`.
4. Объявление метода, инициализирующего компонент `RouterFunction<ServerResponse>`. В данном примере метод вызывается в ходе запуска приложения.
5. Определение `RouterFunction` с использованием методов из `RouterFunctions` и `RequestPredicates`.

В предыдущем примере мы использовали альтернативный способ объявления веб-интерфейса приложения. Он основан на использовании функционального объявления обработчика, позволяющего определить все маршруты в одном месте. Кроме того, такой API позволяет легко писать свои предикаты запросов. Например, следующий пример показывает, как реализовать свой предикат `RequestPredicate` и использовать его в логике маршрутизации.

```

    nest((serverRequest) -> serverRequest.cookies()
        .containsKey("Redirect-Traffic"),
        route(all(), serverRedirectHandler))

```

Здесь мы определили свою короткую реализацию `RouterFunction`, которая переадресует трафик на другой сервер, если в нем присутствует cookie "Redirect-Traffic".

Новый функциональный интерфейс также вводит новый способ обработки запросов и ответов. Например, в следующем примере показана часть реализации `OrderHandler`.

```

class OrderHandler {                                     // (1)
    final OrderRepository orderRepository;              //
    ...

```



```
public Mono<ServerResponse> create(ServerRequest request) { // (2)
    return request
        .bodyToMono(Order.class) // (2.1)
        .flatMap(orderRepository::save) //
        .flatMap(o ->
            ServerResponse.created(URI.create("/orders/" + o.id)) // (2.2)
                .build()) //
        ); //
} //
... //
} //
```

Пояснения к коду.

1. Объявление класса `OrderHandler`. В этом примере мы опустили объявление конструктора, чтобы уделить внимание функциональному API конструирования маршрутов.
2. Объявление метода `create`. Как видите, метод принимает `ServerRequest`, характерный тип запроса для функциональной маршрутизации. Как можно видеть в строке (2.1), `ServerRequest` предлагает API, позволяющий вручную отобразить тело запроса в `Mono` или `Flux`. Кроме того, API дает возможность указать класс, в который требуется отобразить тело запроса. Наконец, функциональное дополнение в `WebFlux` поддерживает API, позволяющий конструировать ответ, используя текущий (fluent) API класса `ServerResponse`.

Как можно заметить, помимо API для объявления маршрутов в функциональном стиле, в нашем распоряжении имеется функциональный API для обработки запроса и ответа.

Несмотря на то что новый API позволяет использовать функциональный стиль объявления обработчиков, его одного недостаточно для создания легковесного веб-приложения. В некоторых случаях полная функциональность экосистемы `Spring` может оказаться избыточной, неоправданно увеличивающей время запуска приложения. Допустим, что мы создаем службу, которая отвечает за сопоставление пользовательских паролей. Обычно такая служба потребляет значительный объем вычислительных ресурсов, хешируя входящие пароли и сопоставляя их с хранимыми контрольными суммами. Единственное, что нужно такой службе, – это интерфейс `PasswordEncoder` из модуля `Spring Security`, позволяющий сравнивать закодированные пароли с исходными текстовыми строками с помощью метода `PasswordEncoder#matches`. Поэтому вся инфраструктура `Spring` с механизмом IoC, обработкой аннотаций и автоматической настройкой оказывается избыточной и замедляет запуск приложения.

К счастью, новый функциональный веб-фреймворк позволяет конструировать веб-приложения без запуска всей инфраструктуры `Spring`. Рассмотрим следующий пример, который демонстрирует, как это сделать.

```

class StandaloneApplication {                                     // (1)

    public static void main(String[] args) {                     // (2)
        HttpHandler httpHandler = RouterFunctions.toHttpHandler( // (2.1)
            routes(new BCryptPasswordEncoder(18))                // (2.2)
        );                                                       //
        ReactorHttpHandlerAdapter reactorHttpHandler =          // (2.3)
            new ReactorHttpHandlerAdapter(httpHandler);          //

        HttpServer.create()                                     // (3)
            .port(8080)                                         // (3.1)
            .handle(reactorHttpHandler)                         // (3.2)
            .bind()                                             // (3.3)
            .flatMap(DisposableChannel::onDispose)              // (3.4)
            .block();                                           //
    }                                                           //

    static RouterFunction<ServerResponse> routes(                // (4)
        PasswordEncoder passwordEncoder                         //
    ) {                                                         //
        return
            route(POST("/check"),                               // (5)
                request -> request                               //
                    .bodyToMono(PasswordDTO.class)               // (5.1)
                    .map(p -> passwordEncoder                     //
                        .matches(p.getRaw(), p.getSecured()))    // (5.2)
                    .flatMap(isMatched -> isMatched              // (5.3)
                        ? ServerResponse
                            .ok()                                 //
                            .build()                              //
                        : ServerResponse
                            .status(HttpStatus.EXPECTATION_FAILED) //
                            .build()                              //
                    )
            );
    }
}

```

Пояснения к коду.

1. Объявление главного класса приложения. Как видите, здесь нет никаких дополнительных аннотаций из Spring Boot.
2. Объявление метода `main` с инициализацией необходимых переменных. В строке (2.2) вызывается метод `routes`, и затем `RouterFunction` преобразуется в `HttpHandler`. В строке (2.3) используется встроенный адаптер `HttpHandler` с именем `ReactorHttpHandlerAdapter`.
3. Создается экземпляр `HttpServer`, являющийся частью Reactor-Netty API. Для настройки сервера мы используем текущий (fluent) API класса `HttpServer`.

В строке (3.1) объявляется порт, добавляется созданный экземпляр `Reactor HttpHandlerAdapter` (строка 3.2), вызовом `bind` в строке (3.3) запускается серверный движок. Наконец, чтобы предотвратить непреднамеренное завершение приложения, блокируем главный поток `Thread` активируем прием события удаления созданного сервера (3.4).

4. Объявление метода `routes`.
5. Логика отображения маршрутов, которая обрабатывает любые запросы `POST`, включающие путь `/check`. Здесь входящий запрос сначала отображается вызовом метода `bodyToMono`. Затем, после преобразования тела, используется экземпляр `PasswordEncoder` для проверки простой текстовой строки с закодированным паролем (в данном случае используется надежный алгоритм `BCrypt` с 18 циклами хеширования, из-за чего на кодирование/сопоставление может потребоваться несколько секунд) (5.2). Наконец, если пароль, введенный пользователем, совпал с хранимым, `ServerResponse` возвращает состояние `OK(200)`, иначе возвращается `EXPECTATION_FAILED(417)`.

Предыдущий пример показывает, как легко получить веб-приложение без необходимости запускать полную инфраструктуру `Spring Framework`. Преимущество веб-приложения в том, что оно имеет намного более короткое время запуска – порядка 700 мс, тогда как запуск того же приложения, но с полной инфраструктурой `Spring Framework` и `Spring Boot`, занимает около 2 секунд (2000 мс), то есть почти в три раза больше.



Обратите внимание, что время запуска у вас может быть иным, но пропорция должна сохраниться.

Итак, используя функциональный стиль объявления маршрутов, мы можем хранить всю конфигурацию маршрутов в одном месте и использовать реактивный подход для обработки входящих запросов. В то же время такой метод предлагает почти ту же гибкость, что и обычный подход на основе аннотаций, с точки зрения доступности параметров, путей и других важных компонентов входящего запроса. Он также помогает избежать запуска всей инфраструктуры `Spring Framework` и обладает той же гибкостью в терминах настройки маршрутов, что может уменьшить время запуска приложения почти в три раза.

Неблокирующие взаимодействия между службами с WebClient

В предыдущих разделах мы рассмотрели базовый дизайн и изменения в новом модуле `Spring WebFlux`, а также познакомились с новыми функциональными подходами, реализованными в `RoutesFunction`. Однако `Spring WebFlux` содержит

и другие новые возможности, в том числе новый неблокирующий HTTP-клиент, который называется *WebClient*.

По сути, *WebClient* является реактивной заменой старого *RestTemplate*. Однако *WebClient* предлагает функциональный API, который лучше соответствует реактивному подходу, и встроенные средства для преобразования типов из *Project Reactor*, таких как *Flux* и *Mono*. Чтобы познакомиться с *WebClient* поближе, рассмотрим следующий пример.

```
WebClient.create("http://localhost/api")           // (1)
    .get()                                           // (2)
    .uri("/users/{id}", userId)                     // (3)
    .retrieve()                                     // (4)
    .bodyToMono(User.class)                         // (5)
    .map(...)                                       // (6)
    .subscribe();                                  //
```

В этом примере мы создаем экземпляр *WebClient*, используя фабричный метод *create* (1). Метод *create* позволяет указать базовый URI, который будет применяться для отправки всех HTTP-запросов. Затем, чтобы создать запрос для отправки удаленному серверу, можно вызвать один из методов *WebClient*, который выглядит как имя HTTP-метода. В предыдущем примере мы использовали метод *WebClient#get* (2). После вызова метода *WebClient#get* получаем экземпляр построителя запроса и с его помощью задаем относительный путь, вызвав метод *uri* (3). Кроме относительного пути можно задать заголовки, блоки данных *cookies* и тело запроса. Однако ради простоты мы опустили эти настройки и сразу перешли к составлению запроса вызовом метода *retrieve* (также можно использовать метод *exchange*). В этом примере мы использовали *retrieve* (4). Данный путь может пригодиться, когда интерес представляет только тело запроса, которое извлекается этим методом и становится доступным для дальнейшей обработки. После подготовки запроса можно использовать один из методов, который поможет преобразовать тело ответа. Здесь мы использовали метод *bodyToMono*, который преобразует входящие данные *User* в *Mono* (5). Наконец, можем сконструировать поток обработки входящего ответа, используя *Reactor API*, и отправить запрос вызовом метода *subscribe*.



WebClient реализует поведение, описанное в стандарте *Reactive Streams*. Это означает, что *WebClient* установит соединение и отправит запрос удаленному серверу только после вызова метода *subscribe*.

Несмотря на то что в большинстве случаев обработке подвергается только тело ответа, иногда бывает необходимо проанализировать возвращаемый код состояния, заголовки или *cookies*. Например, создадим запрос к нашей службе проверки паролей и обработаем код состояния ответа, используя *WebClient API*.

```
class DefaultPasswordVerificationService           // (1)
```

```

implements PasswordVerificationService {                                //
    final WebClient webClient;                                         // (2)
                                                                    //
    public DefaultPasswordVerificationService(                         //
        WebClient.Builder webClientBuilder                             //
    ) {                                                                 //
        this.webClient = webClientBuilder                             // (2.1)
            .baseUrl("http://localhost:8080")                          //
            .build();                                                  //
    }                                                                    //

    @Override                                                           // (3)
    public Mono<Void> check(String raw, String encoded) {              //
        return webClient                                              //
            .post()                                                    // (3.1)
            .uri("/check")                                             //
            .body(BodyInserters.fromPublisher(                         // (3.2)
                Mono.just(new PasswordDTO(raw, encoded)),              //
                PasswordDTO.class)                                     //
            );                                                         //
        .exchange()                                                    // (3.3)
        .flatMap(response -> {                                         // (3.4)
            if (response.statusCode().is2xxSuccessful()) {            // (3.5)
                return Mono.empty();                                    //
            }                                                           //
            else if(resposne.statusCode() == EXPECTATION_FAILED) {    //
                return Mono.error(                                     // (3.6)
                    new BadCredentialsException(...)                  //
                );                                                      //
            }                                                           //
        });                                                            //
        return Mono.error(new IllegalStateException());               //
    }                                                                    //
}

```

Пояснения к коду.

1. Наш класс реализует интерфейс `PasswordVerificationService`.
2. Инициализация экземпляра `WebClient`. Важно отметить, что экземпляр `WebClient` используется для всего класса, поэтому нам не нужно инициализировать новый экземпляр при каждом вызове метода `check`. Такой прием избавляет от необходимости инициализировать новый экземпляр `WebClient` и уменьшает время выполнения метода. Однако в реализации по умолчанию `WebClient` используется `HttpClient` из `Reactor-Netty`, который по умолчанию использует общий пул ресурсов для всех экземпляров `HttpClient`. По этой причине создание нового экземпляра `HttpClient` стоит относительно недорого. После вызова конструктора `DefaultPasswordVerification`

Service начинаем инициализировать `webClient` и используем текущий API построителя, как показано в строке (2.1), для подготовки клиента.

3. Реализация метода `check`. Здесь используем экземпляр `webClient`, чтобы создать POST-запрос (3.1). Также создаем тело, используя метод `body`, и готовимся вставить его, вызвав фабричный метод `BodyInserters#fromPublisher` (3.2). Затем вызываем метод `exchange` (3.3), который возвращает `Mono<ClientResponse>`. Благодаря этому получаем возможность обработать ответ с помощью оператора `flatMap` (3.4). Если пароль успешно прошел проверку (3.5), метод `check` вернет `Mono.empty`. В противном случае, если будет получен код состояния `EXPECTATION_FAILED(417)`, можем вернуть `Mono` с `BadCredentialsException` (3.6).

Как видите, когда требуется проанализировать код состояния, заголовки, cookies и другие внутренние элементы типичного HTTP-ответа, лучше всего использовать метод `exchange`, который возвращает `ClientResponse`.

Как уже упоминалось, по умолчанию `WebClient` использует `HttpClient` из `Reactor-Netty` для выполнения асинхронного и неблокирующего взаимодействия с удаленным сервером. Однако `WebClient` предусматривает простую возможность замены базового HTTP-клиента. Для этого предусмотрена низкоуровневая реактивная абстракция HTTP-соединения в виде `org.springframework.http.client.reactive.ClientHttpConnector`. По умолчанию `WebClient` настроен на использование `ReactorClientHttpConnector`, реализующего интерфейс `ClientHttpConnector`. В версии `Spring WebFlux 5.1` появилась реализация `JettyClientHttpConnector`, которая использует реактивный класс `HttpClient` из библиотеки `Jetty`. Чтобы заменить базовый механизм HTTP-клиента, можно использовать метод `WebClient.Builder#clientConnector` и передать ему желаемый экземпляр, который также может быть существующей или нестандартной пользовательской реализацией.

`ClientHttpConnector` можно использовать не только через удобный слой абстракции, но и непосредственно. Например, его можно использовать для загрузки больших файлов, обработки данных на лету или просто для сканирования потока байтов. Не будем вдаваться в подробное описание `ClientHttpConnector` и оставим любопытным читателям возможность разобраться с ним самостоятельно.

Реактивный WebSocket API

Мы охватили уже большую часть новых особенностей в новом модуле `WebFlux`, однако не коснулись одной из важнейших сторон современной Всемирной паутины – модели потоковых взаимодействий, когда обе стороны, клиент и сервер, обмениваются потоками сообщений. В этом разделе рассмотрим один из самых известных дуплексных протоколов для двусторонней связи, который называется `WebSocket`.

Несмотря на то что поддержка взаимодействий по протоколу WebSocket появилась в Spring Framework еще в начале 2013 года и теоретически предусматривала возможность асинхронной передачи сообщений, фактическая реализация продолжала использовать некоторые блокирующие операции. Например, запись данных в канал ввода/вывода и их чтение продолжали оставаться блокирующими операциями и соответственно оказывали влияние на производительность приложений. По этой причине в модуль WebFlux была добавлена улучшенная версия инфраструктуры поддержки WebSocket.

WebFlux предлагает обе инфраструктуры – для клиента и для сервера. Сначала рассмотрим реализацию поддержки WebSocket на стороне сервера, а потом перейдем к возможностям, доступным на стороне клиента.

Серверный WebSocket API

В качестве центрального интерфейса для обработки соединений WebSocket модуль WebFlux предлагает `WebSocketHandler`. Этот интерфейс имеет метод `handle`, принимающий экземпляр `WebSocketSession`. Класс `WebSocketSession` представляет успешно установленное соединение между клиентом и сервером и обеспечивает доступ к информации, включая информацию о соединении, атрибуты сеанса и входящий поток данных. Чтобы понять, как обращаться с этой информацией, рассмотрим следующий пример, возвращающий отправителю эхо-ответ.

```
class EchoWebSocketHandler implements WebSocketHandler {           // (1)
    @Override                                                       //
    public Mono<Void> handle(WebSocketSession session) {           // (2)
        return session                                              // (3)
            .receive()                                              // (4)
            .map(WebSocketMessage::getPayloadAsText)                // (5)
            .map(tm -> "Echo: " + tm)                               // (6)
            .map(session::textMessage)                              // (7)
            .as(session::send);                                     // (8)
    }                                                                //
}
```

Как можно судить по этому примеру, новый WebSocket API основан на использовании реактивных типов из Project Reactor. В строке (1) мы указываем, что наш класс реализует интерфейс `WebSocketHandler`, и в строке (2) переопределяем метод `handle`. Затем вызываем метод `WebSocketSession#receive` (3), чтобы создать конвейер обработки входящих сообщений `WebSocketMessage`, используя Flux API. `WebSocketMessage` – это класс-обертка вокруг `DataBuffer`, предоставляющий дополнительные возможности, такие как преобразование полезной нагрузки из байтов в текст (5). После извлечения сообщения предваряем его текстом "Echo: " (6), заворачиваем новое сообщение в `WebSocketMessage` и посылаем обратно клиенту, используя метод `WebSocketSession#send`. Метод `send` принимает `Publisher<WebSocketMessage>` и возвращает `Mono<Void>`. Следовательно, исполь-

зую оператор `as` из Reactor API, можем интерпретировать Flux как `Mono<Void>` и использовать `session::send` как функцию преобразования.

Помимо реализации интерфейса `WebSocketHandler` для использования `WebSocket` API на стороне сервера необходимо настроить дополнительные экземпляры `HandlerMapping` и `WebSocketHandlerAdapter`. Как это делается, показано в следующем примере.

```
@Configuration // (1)
public class WebSocketConfiguration { //

    @Bean // (2)
    public HandlerMapping handlerMapping() { //
        SimpleUrlHandlerMapping mapping = //
            new SimpleUrlHandlerMapping(); // (2.1)
        mapping.setUrlMap(Collections.singletonMap( // (2.2)
            "/ws/echo", //
            new EchoWebSocketHandler() //
        )); //
        mapping.setOrder(-1); // (2.3)
        return mapping; //
    } //

    @Bean // (3)
    public HandlerAdapter handlerAdapter() { //
        return new WebSocketHandlerAdapter(); //
    } //
}
```

Пояснения к коду.

1. Объявление класса с аннотацией `@Configuration`.
2. Здесь мы должны объявить и настроить компонент `HandlerMapping`. В строке (2.1) создаем `SimpleUrlHandlerMapping` для настройки отображения путей в `WebSocketHandler`, которая выполняется в строке (2.2). Чтобы `SimpleUrlHandlerMapping` обрабатывал запросы раньше других экземпляров `HandlerMapping`, он должен иметь самый высокий приоритет.
3. Объявление компонента `HandlerAdapter`, который является экземпляром `WebSocketHandlerAdapter`. Здесь экземпляру `WebSocketHandlerAdapter` отводится самая важная роль – он обновляет HTTP-соединение до соединения `WebSocket` и затем вызывает метод `WebSocketHandler#handle`.

Как видите, настройка `WebSocket` API не представляет особой сложности.

Клиентский WebSocket API

В отличие от модуля `WebSocket` (который основан на `WebMVC`), `WebFlux` предлагает также поддержку протокола `WebSocket` на стороне клиента. Чтобы послать за-

прос на создание соединения WebSocket, нужен класс `WebSocketClient`. Для этого класс `WebSocketClient` предлагает два главных метода, как показано в примере ниже.

```
public interface WebSocketClient {
    Mono<Void> execute(
        URI url,
        WebSocketHandler handler
    );
    Mono<Void> execute(
        URI url,
        HttpHeaders headers,
        WebSocketHandler handler
    );
}
```

Как видите, для получения сообщений и отправки ответов серверу `WebSocketClient` использует тот же интерфейс `WebSocketHandler`. Существует несколько реализаций `WebSocketClient`, ориентированных на работу с конкретным сервером, таких как `TomcatWebSocketClient` или `JettyWebSocketClient`. В следующем примере используется реализация `ReactorNettyWebSocketClient`.

```
WebSocketClient client = new ReactorNettyWebSocketClient();

client.execute(
    URI.create("http://localhost:8080/ws/echo"),
    session -> Flux
        .interval(Duration.ofMillis(100))
        .map(String::valueOf)
        .map(session::textMessage)
        .as(session::send)
);
```

Здесь показано, как с помощью `ReactorNettyWebSocketClient` установить WebSocket-соединение и начать периодическую отправку сообщений серверу.

Сравнение WebFlux WebSocket и Spring WebSocket

Читатели, знакомые с модулем `WebSocket` на основе сервлетов, могут заметить, что оба модуля имеют много общего. Тем не менее у них есть немало отличий. Как вы помните, основной недостаток модуля `Spring WebSocket` заключается в блокирующих операциях ввода/вывода, тогда как `Spring WebFlux` предлагает неблокирующие операции чтения/записи. Кроме того, модуль `WebFlux` имеет улучшенную абстракцию потоковых взаимодействий, основанную на стандарте `Reactive Streams` и типах из `Project Reactor`. Интерфейс `WebSocketHandler` из старого модуля `WebSocket` позволяет обрабатывать сообщения только по одному. Метод

WebSocketSession#sendMessage допускает отправку сообщений лишь синхронным способом.

Впрочем, интеграция нового модуля Spring WebFlux с WebSocket тоже имеет свои недостатки. Одной из важнейших черт старого модуля Spring WebSocket была хорошая интеграция с модулем Spring Messaging, которая позволяла использовать аннотацию @MessageMapping для объявления конечной точки WebSocket. Ниже приводится простой пример использования аннотации из Spring Messaging со старым WebSocket API на основе WebMVC.

```
@Controller
public class GreetingController {

    @MessageMapping("/hello")
    @SendTo("/topic/greetings")
    public Greeting greeting(HelloMessage message) {
        return new Greeting("Hello, " + message.getName() + "!");
    }
}
```

Здесь показано, как можно было использовать модуль Spring Messaging для объявления конечной точки WebSocket. К сожалению, такая возможность отсутствует в интеграции WebFlux с WebSocket, поэтому для объявления сложных обработчиков мы вынуждены определять свою собственную инфраструктуру.

В главе 8 «Масштабирование с Cloud Streams» мы расскажем о еще одной мощной абстракции двустороннего обмена сообщениями между клиентом и сервером, которую можно использовать для реализации простых взаимодействий между сервером и браузером.

Реактивный поток SSE и легковесная замена WebSocket

Наряду с тяжеловесным протоколом WebSocket стандарт HTML 5 предлагает новый способ создания статических (в данном случае полудуплексных) соединений, когда сервер получает возможность посылать события. Этот способ решает похожие проблемы, что и WebSocket. Например, можно объявить **поток серверных событий (Server-Sent Events, SSE)**, используя ту же модель программирования на основе аннотаций, но вернуть бесконечный поток объектов ServerSentEvent, как показано в примере ниже.

```
@RestController                                     // (1)
@RequestMapping("/sse/stocks")                       //
class StocksController {                             //
    final Map<String, StocksService> stocksServiceMap; //
    ...                                              //
}
```

```

@GetMapping                                                                    // (2)
public Flux<ServerSentEvent<?>> streamStocks() {                               // (2.1)
    return Flux                                                                //
        .fromIterable(stocksServiceMap.values())                             //
        .flatMap(StocksService::stream)                                       // (2.2)
        .<ServerSentEvent<?>>map(item ->                                       //
            ServerSentEvent                                                    // (2.3)
                .builder(item)                                                 // (2.4)
                .event("StockItem")                                           // (2.5)
                .id(item.getId())                                              // (2.6)
                .build())                                                       //
        )                                                                      //
        .startWith(                                                           // (2.7)
            ServerSentEvent                                                    //
                .builder()                                                      //
                .event("Stocks")                                               // (2.8)
                .data(stocksServiceMap.keySet())                             // (2.9)
                .build())                                                       //
        );                                                                    //
}
}

```

Пояснения к коду.

1. Объявление класса `@RestController`. Чтобы упростить код, мы опустили разделы с объявлением конструктора и инициализацией полей.
2. Объявление метода-обработчика, отмеченного уже знакомой аннотацией `@GetMapping`. Как можно видеть в строке (2.1), метод `streamStocks` возвращает поток `Flux` экземпляров `ServerSentEvent`, то есть текущий обработчик способен обрабатывать поток событий. Затем мы объединяем все доступные источники информации и пересылаем изменения клиенту (2.2). После этого преобразуем каждый `StockItem` в `ServerSentEvent` (2.3), используя статический метод `builder` (2.4). Для настройки экземпляра `ServerSentEvent` передаем строителю параметры: идентификатор события (2.6) и его имя (2.5), что позволит клиенту различать события. Кроме того, в строке (2.7) мы запускаем поток данных `Flux` с конкретным экземпляром `ServerSentEvent` (2.8), который объявляет клиенту доступные каналы информации (2.9).

Как можно заметить в этом примере, *Spring WebFlux* позволяет отобразить потоковую природу реактивного типа `Flux` и послать клиенту бесконечный поток событий. Кроме того, для потоковой передачи SSE не требуется изменять API или использовать дополнительные абстракции. Достаточно объявить конкретный возвращаемый тип, чтобы помочь фреймворку понять, как поступить с ответом. Нам также не нужно объявлять поток `Flux` экземпляров `ServerSentEvent`, вместо этого можно просто указать тип контента, как показано в следующем примере.

```
@GetMapping(produces = "text/event-stream")
public Flux<StockItem> streamStocks() {
    ...
}
```

В данном случае фреймворк WebFlux завернет каждый элемент потока данных в `ServerSentEvent`.

Как видите, основное преимущество способа на основе `ServerSentEvent` заключается в отсутствии необходимости писать дополнительный шаблонный код для настройки такой модели потоковой передачи данных, который мы вынуждены писать, когда используем WebFlux-версию `WebSocket`. Это связано с тем, что SSE – простая абстракция поверх HTTP, не требующая переключения протоколов и специальной настройки сервера.

Также в предыдущем примере можно заметить, что SSE можно настроить с использованием традиционной комбинации аннотаций `@RestController` и `@XXMapping`, тогда как при использовании `WebSocket` приходится определять свою конфигурацию преобразования сообщений, например вручную выбирать конкретный протокол обмена сообщениями. Для SSE модуль Spring WebFlux предлагает те же конфигурации преобразования сообщений, что и типичный контроллер REST.

Однако SSE не поддерживает двоичного кодирования и требует, чтобы все события имели кодировку UTF-8. Это означает, что `WebSocket` позволяет обмениваться более короткими сообщениями и уменьшить трафик между клиентом и сервером, обеспечивая тем самым более низкую задержку.

Подводя итог, отметим, что в целом SSE является хорошей альтернативой протоколу `WebSocket`. Так как SSE представляет абстракцию поверх HTTP, WebFlux поддерживает те же декларативные и функциональные конфигурации конечных точек и преобразования сообщений, что и обычные контроллеры REST.



Узнать больше о достоинствах и недостатках SSE и о сравнительных характеристиках с `WebSocket` можно в статье <https://stackoverflow.com/a/5326159/4891253>.

Реактивные механизмы шаблонов

Наряду с обычными функциями API существенный интерес вызывают средства создания пользовательского интерфейса современных веб-приложений. В наши дни пользовательский интерфейс веб-приложений основан на сложных механизмах отображения с привлечением JavaScript, и в большинстве случаев разработчики предпочитают использовать механизмы отображения, действующие на стороне клиента. Несмотря на это, во многих корпоративных приложениях все еще

используются серверные технологии отображения. Web MVC включает поддержку разных технологий, таких как JSP, JSTL, FreeMarker, Groovy Markup, Thymeleaf, Apache Tiles и др. К сожалению, в Spring 5.x и в модуле WebFlux поддержка многих из них, в том числе Apache Velocity, исчезла.

Тем не менее в Spring WebFlux имеются те же самые приемы визуализации представлений, что и в Web MVC. Следующий пример демонстрирует знакомый способ определения представления для визуализации.

```
@RequestMapping("/")
public String index() {
    return "index";
}
```

Метод `index` в предыдущем примере возвращает экземпляр `String` с именем представления. За кулисами фреймворк отыщет это представление в настроенной папке и отобразит его с использованием соответствующего механизма шаблонов.

По умолчанию WebFlux поддерживает только механизм FreeMarker, действующий на стороне сервера. Однако для нас важно совсем другое – поддерживает ли процесс отображения шаблонов реактивный подход? Чтобы ответить на этот вопрос, рассмотрим случай отображения небольшого списка музыкальных произведений.

```
@RequestMapping("/play-list-view")
public Mono<String> getPlaylist(final Model model) {                // (1)
    final Flux<Song> playlistStream = ...;                          // (2)
    return playlistStream                                          //
        .collectList()                                           // (3)
        .doOnNext(list -> model.addAttribute("playList", list)) // (4)
        .then(Mono.just("freemarker/play-list-view"));           // (5)
}
```

Как показано в этом примере, мы используем реактивный тип `Mono<String>` (1), чтобы асинхронно вернуть имя представления. Кроме того, наш шаблон включает заполнитель, `dataSource`, который следует заменить списком `Song` (2). Часто для передачи контекстной информации определяется модель `Model` (1), в которую добавляется требуемый атрибут (4). К сожалению, FreeMarker не поддерживает реактивное и неблокирующее отображение данных, поэтому мы должны собрать все произведения в список и вставить его в `Model`. Наконец, после сбора всех данных и записи их в `Model` можем вернуть имя представления и начать его отображение.

К сожалению, отображение шаблонов таким образом, как происходит здесь, требует значительных вычислительных ресурсов. При наличии большого набора данных это может привести к расходованию некоторого времени и памяти. К счастью,

сообщество Thymeleaf решило поддержать реактивный модуль *WebFlux* и реализовать асинхронное и потоковое отображения шаблонов. Thymeleaf предлагает возможности, аналогичные возможностям *FreeMarker*, и позволяет писать идентичный код для отображения пользовательских интерфейсов. Также Thymeleaf дает возможность использовать внутри шаблонов реактивные типы в качестве источников данных и отображать часть шаблона, когда в потоке данных появляется новый элемент. В следующем примере показано, как использовать реактивные потоки с Thymeleaf в ходе обработки запроса.

```
@RequestMapping("/play-list-view")
public String view(final Model model) {
    final Flux<Song> playlistStream = ...;
    model.addAttribute(
        "playList",
        new ReactiveDataDriverContextVariable(playlistStream, 1, 1)
    );
    return "thymeleaf/play-list-view";
}
```

Этот пример вводит новый тип данных *ReactiveDataDriverContextVariable*, принимающий реактивные типы, такие как *Publisher*, *Flux*, *Mono*, *Observable* и другие, поддерживаемые классом *ReactiveAdapterRegistry*.

Несмотря на то что для поддержки реактивности требуется дополнительный класс, обертывающий потоки данных, на стороне шаблона не требуется никаких изменений. Следующий пример показывает, что с реактивными потоками можно работать как с обычными коллекциями.

```
<!DOCTYPE html>                                <!-- (1) -->
<html>
    ...
    <body>
        ...
        <table>                                    <!-- (2) -->
            <thead>
                ...                                <!-- (3) -->
            </thead>
            <tbody>                                <!-- (4) -->
                <tr th:each="e : ${playList}">      <!-- (5) -->
                    <td th:text="${e.id}">...</td>
                    <td th:text="${e.name}">...</td>
                    <td th:text="${e.artist}">...</td>
                    <td th:text="${e.album}">...</td>
                </tr>
            </tbody>
        </table>
```

```
</body>  
</html>
```

Этот код демонстрирует, как использовать разметку шаблона Thymeleaf, который включает типичное объявление документа HTML (1). Он отображает таблицу (2) с несколькими заголовками столбцов (3) и телом (4). Она заполняется строками, конструируемыми из элементов Song в playList.

Самое большое преимущество заключается в том, что механизм отображения Thymeleaf начинает потоковую передачу данных клиенту, не дожидаясь получения последнего элемента. Более того, он поддерживает отображение бесконечного потока элементов. Это стало возможным благодаря добавлению поддержки Transfer-Encoding: chunked. Вместо создания полного отображения всего шаблона в памяти Thymeleaf сначала отображает доступные части, а затем асинхронно посылает оставшиеся части шаблона по мере появления новых элементов.

К сожалению, на момент написания этих строк Thymeleaf поддерживал для каждого шаблона только один источник реактивных данных. Тем не менее он позволяет вернуть клиенту первый фрагмент данных намного раньше, чем это делают классические механизмы отображения, которые требуют наличия всего набора данных, и уменьшает задержку между запросом и первой порцией возвращаемых данных, что улучшает общее восприятие пользователя.

Реактивная безопасность

Одним из важнейших аспектов современных веб-приложений является безопасность. С первых лет существования фреймворка Spring Web в его состав входил вспомогательный модуль Spring Security. Он позволяет обезопасить веб-приложение и естественным образом вписывается в существующую инфраструктуру Spring Web, предлагая возможность фильтрации до вызова любого контроллера или обработчика. Долгие годы модуль Spring Security был тесно связан с инфраструктурой Web MVC и использовал только абстракцию Filter из Servlet API.

К счастью, ситуация изменилась с появлением реактивного модуля WebFlux. Для поддержки реактивных и неблокирующих взаимодействий между компонентами Spring Security предлагает реализацию совершенно нового реактивного стека, которая использует новую инфраструктуру WebFilter и в значительной степени опирается на поддержку *контекста* из Project Reactor.

Реактивный доступ к SecurityContext

Для доступа к SecurityContext в новом реактивном модуле Spring Security имеется новый класс ReactiveSecurityContextHolder.

Он предлагает реактивный доступ к текущему контексту SecurityContext посредством статического метода getContext, который возвращает Mono<Security

Context>. Это означает, что обратиться к SecurityContext в приложении можно так, как показано ниже.

```
@RestController // (1)
@RequestMapping("/api/v1") //
public class SecuredProfileController { //
    @GetMapping("/profiles") // (2)
    @PreAuthorize("hasRole(USER)") // (2.1)
    public Mono<Profile> getProfile() { // (2.2)
        return ReactiveSecurityContextHolder // (2.3)
            .getContext() // (2.4)
            .map(SecurityContext::getAuthentication) //
            .flatMap(auth -> //
                profileService.getByUser(auth.getName()) // (2.5)
            ); //
    } //
}
```

Пояснения к коду.

1. Объявление класса контроллера REST, которому передаются запросы с "/api/v1".
2. Объявление метода-обработчика getProfile. Как видите, метод возвращает реактивный тип Mono, обеспечивающий реактивный доступ к данным (2.2). Далее, чтобы получить доступ к текущему SecurityContext, вызывается ReactiveSecurityContextHolder.getContext() (2.3) и (2.4). Наконец, если SecurityContext присутствует, вызывается flatMap, и мы получаем доступ к профилю пользователя, как показано в строке (2.5). Кроме того, метод снабжен аннотацией @PreAuthorize, которая в данном случае проверяет наличие требуемой роли в доступном экземпляре Authentication. Обратите внимание, что если метод возвращает реактивный тип, его вызов будет отложен до момента, когда будет выполнена аутентификация и получены необходимые привилегии.

Как можно заметить, API нового реактивного держателя контекста похож на имеющийся в синхронном аналоге. Более того, с новым поколением Spring Security можно использовать те же аннотации для проверки привилегий.

Внутренне ReactiveSecurityContextHolder опирается на Context API из Reactor. Информация о текущем зарегистрировавшемся пользователе хранится в экземпляре интерфейса Context. Следующий пример показывает, как работает ReactiveSecurityContextHolder.

```
static final Class<?> SECURITY_CONTEXT_KEY = SecurityContext.class;
...
public static Mono<SecurityContext> getContext() {
    return Mono.subscriberContext()
```



```
.filter(c -> c.hasKey(SEcurityContextKey))  
.flatMap(c -> c.<Mono<SecurityContext>>get(SEcurityContextKey));  
}
```

Как рассказывалось в главе 4 «*Project Reactor – основа реактивных приложений*», чтобы получить доступ к внутреннему экземпляру Context, можно использовать специальный оператор `subscriberContext` реактивного типа Mono. После получения контекста вызывается его метод `filter` и проверяется наличие требуемого ключа. Этот ключ хранит экземпляр потока Mono с объектом `SecurityContext`, следовательно, доступ к текущему `SecurityContext` осуществляется реактивным способом. Извлечение экземпляра `SecurityContext`, хранящегося, например, в базе данных, происходит, только когда кто-то подпишется на этот поток Mono.

Несмотря на то что API класса `ReactiveSecurityContextHolder` выглядит знакомым, в нем множество подводных камней. Например, можно по ошибке последовать за привычкой, выработавшейся использованием `SecurityContextHolder`, и реализовать типичное взаимодействие следующим образом:

```
ReactiveSecurityContextHolder  
    .getContext()  
    .map(SecurityContext::getAuthentication)  
    .block();
```

Привыкнув извлекать `SecurityContext` из `ThreadLocal`, можно по ошибке попытаться сделать то же самое с `ReactiveSecurityContextHolder`, как показано в этом примере. К сожалению, после вызова `getContext` и подписки с помощью `block` мы получим пустой контекст в потоке данных. Поэтому, когда класс `ReactiveSecurityContextHolder` попытается извлечь внутренний экземпляр Context, он не найдет доступного экземпляра `SecurityContext`.

Встает вопрос: как правильно настроить контекст Context и сделать его доступным при правильном подключении потоков, как показано в начале раздела? Ответ заключается в новом `ReactorContextWebFilter` из модуля Spring Security пятого поколения. Во время вызова `ReactorContextWebFilter` дает возможность получить экземпляр Context с помощью метода `subscriberContext`. Кроме того, разрешение `SecurityContext` выполняется с помощью класса `ServerSecurityContextRepository`, имеющего два метода: `save` и `load`.

```
interface ServerSecurityContextRepository {  
    Mono<Void> save(ServerWebExchange exchange, SecurityContext context);  
    Mono<SecurityContext> load(ServerWebExchange exchange);  
}
```

Как показано в этом примере, метод `save` позволяет связать `SecurityContext` с конкретным `ServerWebExchange` и затем восстановить его с помощью метода `load` из входящего пользовательского запроса, прикрепленного к `ServerWebExchange`.

Основным достоинством модуля Spring Security нового поколения является полноценная поддержка реактивного доступа к `SecurityContext`. В данном случае реактивный доступ предполагает возможность хранения `SecurityContext` в базе данных и его извлечение без выполнения блокирующих операций. Фактическое извлечение контекста из хранилища откладывается до момента, когда будет выполнена подписка на `ReactiveSecurityContextHolder.getContext()`. Наконец, механизм передачи `SecurityContext` позволяет легко конструировать сложные процессы обработки потоков, не беспокоясь о распространенной проблеме передачи `ThreadLocal` между экземплярами `Thread`.

Использование реактивной безопасности

Теперь давайте посмотрим, насколько сложно внедрить поддержку безопасности в реактивное веб-приложение. К счастью, для настройки безопасности в современном приложении на основе *WebFlux* достаточно объявить несколько компонентов. Ниже приводится справочный пример, показывающий, как это сделать.

```
@SpringBootApplication                                // (1)
@EnableReactiveMethodSecurity                          // (1.1)
public class SecurityConfiguration {                  //

    @Bean                                              // (2)
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http                      // (2.1)
    ) {                                              //
        return http                                  // (2.2)
            .formLogin()                             //
            .and()                                    //
            .authorizeExchange()                      //
            .anyExchange().authenticated()           //
            .and()                                    //
            .build();                                // (2.3)
    }                                              //

    @Bean                                              // (3)
    public ReactiveUserDetailsService userDetailsService() {
        UserDetails user =                           //
            User.withUsername("user")                 // (3.1)
                .withDefaultPasswordEncoder()         //
                .password("password")                 //
                .roles("USER", "ADMIN")               //
                .build();                             //
        return new MapReactiveUserDetailsService(user); // (3.2)
    }
}
```

Пояснения к коду.

1. Объявление класса с настройками. Здесь, чтобы включить конкретный аннотированный `MethodInterceptor`, нужно добавить аннотацию `@EnableReactiveMethodSecurity`, которая импортирует необходимые для этого конфигурации (1.1).
2. Определяется конфигурация для компонента `SecurityWebFilterChain`. Для настройки нужного компонента Spring Security предлагает `ServerHttpSecurity` – построитель (2.3) с текущим API (2.2).
3. Определяется конфигурация для компонента `ReactiveUserDetailsService`. Для аутентификации пользователя с настройками Spring Security по умолчанию мы должны передать реализацию `ReactiveUserDetailsService`. Для простоты в данном примере используем реализацию интерфейса в памяти (3.2) и настраиваем тестового пользователя (3.1) для входа в систему.

Как можно заметить в предыдущем коде, в общем и целом настройка Spring Security напоминает то, что мы видели раньше. Соответственно, переход на эту версию не потребует больших усилий.

Поддержка реактивности в новом поколении Spring Security позволяет создавать хорошо защищенные веб-приложения с минимальными усилиями на настройку инфраструктуры.

Взаимодействия с другими реактивными библиотеками

Несмотря на то что в роли основного строительного блока модуль WebFlux использует Project Reactor 3, он поддерживает и другие реактивные библиотеки. Для поддержки совместимости библиотек большинство операций в WebFlux выполняется с использованием интерфейсов, определяемых стандартом Reactive Streams. Благодаря этому мы можем легко заменить Reactor 3 на RxJava 2 или Akka Streams.

```
import io.reactivex.Observable; // (1)
... //
@RestController // (2)
class AlbumsController { //
    final ReactiveAdapterRegistry adapterRegistry; // (2.1)
    ... //

    @GetMapping("/songs") // (3)
    public Observable<Song> findAlbumByArtists( // (3.1)
        Flux<Artist> artistsFlux // (3.2)
    ) {
        Observable<Artist> observable = adapterRegistry // (4)
```

```
        .getAdapter(Observable.class)                //  
        .fromPublisher(artistsFlux);                 //  
        Observable<Song> albomsObservable = ...;      // (4.1)  
                                                    //  
        return albomsObservable;                    // (4.2)  
    }  
}
```

Пояснения к коду.

1. Объявление импорта, демонстрирующее, что в данном примере мы импортируем `Observable` из `RxJava` 2.
2. Класс `AlbomsController` с аннотацией `@RestController`. Здесь также объявляется поле типа `ReactiveAdapterRegistry`, используемое далее в этом примере.
3. Объявление метода `findAlbumByArtists` для обработки запросов. Как видите, метод `findAlbumByArtists` принимает издателя `Publisher` типа `Flux<Artist>` (3.2) и возвращает `Observable<Song>` (3.1).
4. Преобразование `artistsFlux` в `Observable<Artist>`, выполнение бизнес-логики (4.1) и возврат результата.

Предыдущий пример показывает, как можно реализовать реактивные взаимодействия с использованием реактивных типов из `RxJava` и из `Project Reactor`. Как рассказывалось в главе 5 «Добавление реактивности с помощью *Spring Boot 2*», преобразование реактивных типов является частью модуля `Spring Core` и поддерживается посредством `org.springframework.core.ReactiveAdapterRegistry` и `org.springframework.core.ReactiveAdapter`. Эти классы позволяют выполнять преобразование в класс `Publisher`, определяемый стандартом `Reactive Streams`, и обратно. Благодаря такой поддержке можем использовать практически любые реактивные библиотеки, а не только `Project Reactor`.

Сравнение *WebFlux* и *Web MVC*

В предыдущих разделах мы кратко перечислили основные компоненты, включенные в новый модуль `Spring WebFlux`, познакомились с новыми возможностями, появившимися в нем, и узнали, как ими пользоваться.

Однако даже понимая, как пользоваться новым API для создания веб-приложений, пока мы не знаем, почему новый `WebFlux` считается лучше `Web MVC`. Было бы интересно осознать основные преимущества `WebFlux`.

Для этого нам придется немного углубиться в теоретические основы конструирования веб-серверов, понять, какие характеристики важны для быстрого веб-сер-

вера, и рассмотреть аспекты, влияющие на его производительность. В следующих разделах проанализируем характеристики современных веб-серверов, посмотрим, что может вызвать деградацию производительности, и поразмышляем над тем, как этого избежать.

Законы сравнения фреймворков

Перед тем как продолжить, определим характеристики системы, которая будет использоваться в сравнительном анализе. Основными показателями для большинства веб-приложений являются пропускная способность, задержка, а также потребление памяти. В настоящее время Всемирная паутина предъявляет совсем другие требования, чем на начальном этапе развития. Прежде компьютеры работали последовательно. Пользователей вполне удовлетворяло простое и статичное содержимое, а общая нагрузка на систему была достаточно низкой. К числу основных операций относилось создание разметки HTML и простейшие вычисления. С вычислениями вполне справлялся единственный процессор, и для веб-приложения хватало одного сервера.

Спустя годы правила игры изменились. Счет пользователей Всемирной паутины пошел на миллиарды, а содержимое страниц стало динамичным, а иногда даже обновляемым в режиме реального времени. Требования к пропускной способности и задержке существенно изменились. Наши веб-приложения начали приобретать распределенный характер. Понимание принципов масштабирования веб-приложений приобрело особую значимость. Важный вопрос – как число параллельно выполняющихся рабочих потоков влияет на задержку или пропускную способность?

Закон Литтла

Ответ на этот вопрос дает закон Литтла. Он объясняет, как рассчитать, сколько запросов должно обрабатываться одновременно (или, проще говоря, каково количество одновременно действующих рабочих потоков выполнения), чтобы обеспечить требуемую пропускную способность при конкретном уровне задержки. Иными словами, используя эту формулу, можно рассчитать емкость системы, то есть количество компьютеров, узлов или экземпляров веб-приложения, действующих параллельно, чтобы обслужить требуемое число пользовательских запросов в секунду при неизменном времени отклика:

$$N = X \times R.$$

Эта формула читается так: *среднее количество запросов (N), находящихся в системе или в очереди (то есть обрабатываемых одновременно), равно пропускной способности (X) – числу пользователей в секунду, – умноженной на среднее время отклика или задержки (R).*

То есть, чтобы среднее время отклика R системы составляло 0,2 секунды и она обрабатывала до $X = 100$ запросов в секунду, система должна иметь возможность одновременно (то есть параллельно) обслуживать до 20 запросов (пользователей). Для этого нужна одна машина с 20 рабочими потоками выполнения или 20 машин с одним рабочим потоком выполнения на каждой. Идеально, когда рабочие потоки не пересекаются и отсутствуют одновременные запросы. Этот случай изображен на рис. 6.3.

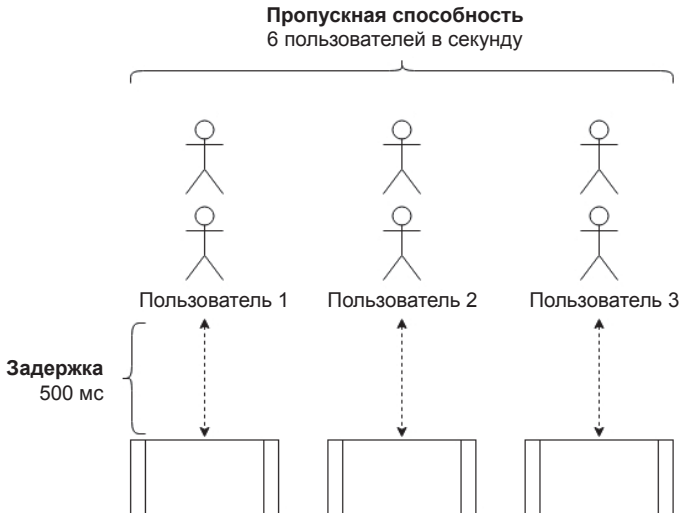


Рис. 6.3. Идеальный случай параллельной обработки

Как показано на рис. 6.3, в системе действует три рабочих потока выполнения, которые могут обслуживать до шести пользователей или запросов в секунду. В этом случае обслуживание пользователей *равномерно распределено* по рабочим потокам выполнения, а это значит, что нет нужды использовать дополнительный механизм для координации в выборе рабочего потока.

Однако такой идеальный случай очень редко встречается на практике, потому что любая система, как, например, веб-приложение, требует возможности конкурентного доступа к общим ресурсам, таким как процессор или память. Соответственно, возникает необходимость вносить поправки в общую пропускную способность, которые описываются *законом Амдала* и дополняющим его *универсальным законом масштабируемости*.

Закон Амдала

Первый из этих законов касается влияния *последовательного доступа* на среднее время отклика (или задержки), а следовательно, и на пропускную способность. Несмотря на то что у нас всегда есть желание распараллелить работу, может наступить момент, когда это невозможно и ее приходится выполнять последова-

тельно. Это может иметь место, когда есть некоторый координатор заданий, или оператор агрегирования или свертки в реактивном конвейере, что означает необходимость присоединения ко всем потокам выполнения. Или это может быть фрагмент кода, способный работать только последовательно. В больших системах на основе микросервисов это может быть балансировщик нагрузки или система согласования. Как следствие мы должны обратиться к закону Амдала, чтобы рассчитать изменение пропускной способности, используя следующую формулу:

$$X(N) = \frac{X(1) \times N}{1 + \sigma \times (N - 1)}.$$

В этой формуле **$X(1)$** представляет *первоначальную пропускную способность*, **N** – коэффициент распараллеливания, или количество рабочих потоков выполнения, **σ** – коэффициент конкуренции (иногда его также называют коэффициентом сериализации) или, другими словами, доля общего времени, потраченная на выполнение кода, который не может выполняться параллельно.

Если сделать простейший расчет и построить график зависимости пропускной способности от возможности распараллеливания заданий с некоторым произвольно выбранным коэффициентом конкуренции, $\sigma = 0,03$ и начальной пропускной способностью **$X(1) = 50$** запросов в секунду в диапазоне распараллеливания $N = 0..500$, получим кривую, изображенную на рис. 6.4.

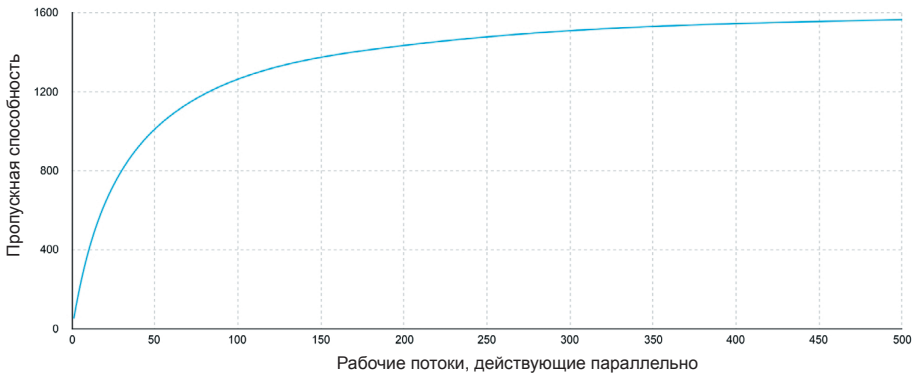


Рис. 6.4. Изменение пропускной способности в зависимости от степени распараллеливания

Судя по графику на рис. 6.4, с увеличением параллельно выполняющихся рабочих потоков пропускная способность сначала быстро растет, а затем ее рост замедляется все больше и больше. Наконец, рост пропускной способности останавливается. Закон Амдала утверждает, что общее распараллеливание работы не дает линейного увеличения пропускной способности, потому что невозможно обрабатывать результаты быстрее, чем фрагмент кода, выполняющий обработку последовательно. С точки зрения масштабирования обычного веб-приложения это означает, что увеличение числа ядер или узлов в системе не даст положительного эффекта, если имеется единственная точка координации или обработки, которая

не может работать быстрее. Более того, мы просто потеряем деньги на поддержку избыточного количества машин, и мизерное увеличение общей пропускной способности не оправдывает затраченные средства.

Рассматривая график на рис. 6.4, можно заметить, что изменение пропускной способности зависит от степени распараллеливания. Однако в большинстве случаев также важно понимать, как изменяется задержка с увеличением степени распараллеливания. Для этого можно объединить уравнения, описывающие закон Литтла и закон Амдала. Оба уравнения содержат член, описывающий пропускную способность (X). Следовательно, мы должны переписать закон Амдала, чтобы объединить обе формулы:

$$X(N) = \frac{N}{R}.$$

Если теперь подставить правую часть из уравнения Амдала вместо $X(N)$, получим

$$\frac{N}{R} = \frac{X(1) \times N}{1 + \sigma \times (N - 1)}.$$

А теперь выведем отсюда задержку (R):

$$\frac{1}{R} = \frac{X(1) \times N}{(1 + \sigma \times (N - 1)) \times N};$$

$$R = \frac{1 + \sigma \times (N - 1)}{X(1)}.$$

Из этой формулы можно заключить, что в общем случае задержка растет линейно. На рис. 6.5 показана кривая роста задержки в зависимости от степени распараллеливания.

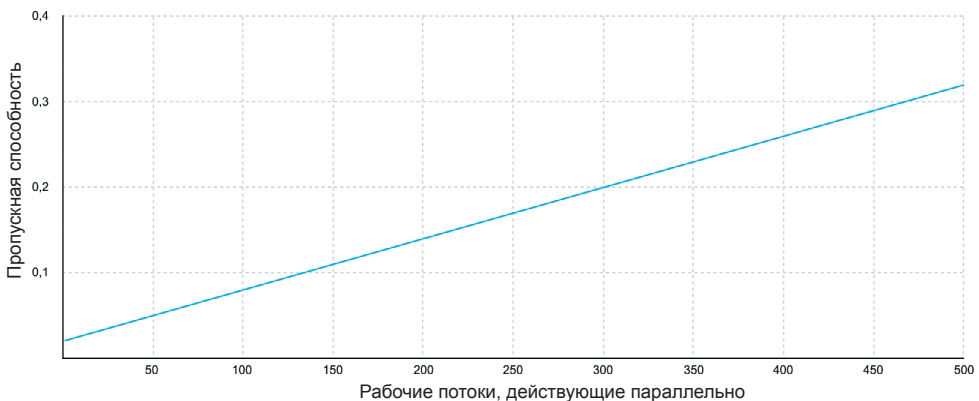


Рис. 6.5. Линейный рост задержки в зависимости от степени распараллеливания

То есть с увеличением числа рабочих потоков, выполняющихся параллельно, время отклика уменьшается.

Наконец, согласно закону Амдала, система с параллельным выполнением задания всегда имеет точки последовательного выполнения, что приводит к дополнительным издержкам и не позволяет достичь более высокой пропускной способности простым увеличением степени распараллеливания. Диаграмма на рис. 6.6 иллюстрирует эту систему.

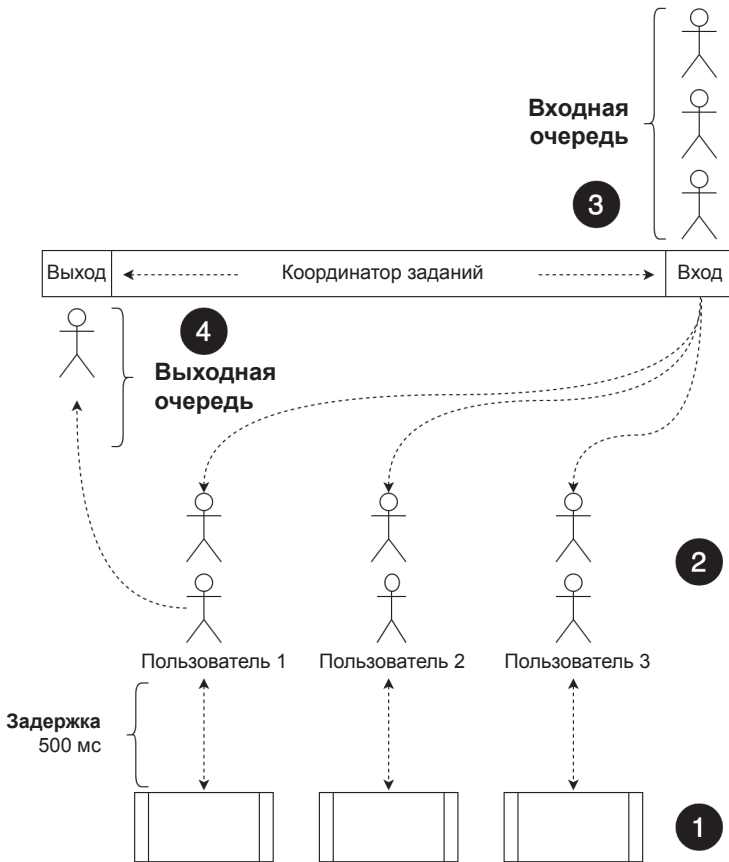


Рис. 6.6. Пример системы с параллельной обработкой заданий, соответствующей закону Амдала

Пояснения к рис. 6.6.

1. Представление задания. Обратите внимание, что даже здесь код нельзя разбить на меньшие подзадачи, выполняемые независимо, что также следует рассматривать как точку с последовательным выполнением.
2. Представление пользователей или запросов в очереди.
3. Очередь пользователей или запросов перед передачей для обработки рабочим потоком. Точкой последовательного выполнения здесь является координация и связывание обработки запроса с определенным рабочим потоком выполнения.

4. Координация запросов может осуществляться в обоих направлениях. В этой точке координатор может выполнять некоторые действия перед отправкой ответов пользователям.

Таким образом, закон Амдала утверждает, что любая система имеет свои узкие места, поэтому мы не можем обслужить больше пользователей или уменьшить задержку.

Универсальный закон масштабируемости

Несмотря на то что масштабируемость любой системы объясняется законом Амдала, реальные приложения показывают совершенно разные результаты масштабируемости. После тщательных исследований в этой области Нил Гюнтер (Neil Gunther) обнаружил, что кроме точек последовательного выполнения существует еще один важный аспект, который называется **несогласованностью**.



Нил Гюнтер (Neil Gunther) – исследователь компьютерных информационных систем, получивший всемирную известность за разработку открытого программного обеспечения Pretty Damn Quick для моделирования производительности и подхода Guerrilla к планированию и анализу производительности. За дополнительной информацией обращайтесь по ссылке <http://www.perfdynamics.com/Bio/njg.html>.

Несогласованность – обычное явление в системе с общими ресурсами. Например, с точки зрения стандартного веб-приложения на Java несогласованность проявляется в виде хаотичного доступа потоков выполнения Thread к ресурсам, таким как процессор (CPU). Вся модель потоков выполнения в Java далека от идеала. Если число экземпляров Thread намного превышает фактическое число процессоров, экземпляры Thread вступают в конкурентную борьбу за обладание процессором и выполнение вычислений. Для их согласования и устранения их избыточной координации приходится прикладывать дополнительные усилия. Каждое обращение к общей памяти из потока выполнения может потребовать дополнительной синхронизации и ухудшить пропускную способность и величину задержки приложения.

Чтобы объяснить такое поведение системы, **универсальный закон масштабируемости (Universal Scalability Law, USL)**, дополняющий закон Амдала, предлагает следующую формулу для вычисления изменения пропускной способности в зависимости от степени распараллеливания:

$$X(N) = \frac{X(1) \times N}{1 + \sigma \times (N - 1) + \kappa \times N \times (N - 1)} .$$

В предыдущей формуле появился новый коэффициент – коэффициент согласования (**κ**). Самое примечательное, что согласно этому закону пропускная способность **$X(N)$** находится в обратной квадратичной зависимости от степени распараллеливания **N** .

Чтобы понять фатальное влияние этой зависимости, рассмотрим график на рис. 6.7, построенный с использованием тех же начальных условий: начальная пропускная способность $X(1) = 50$, коэффициент конкуренции $\sigma = 0,03$ и коэффициент согласования $k = 0,00007$.

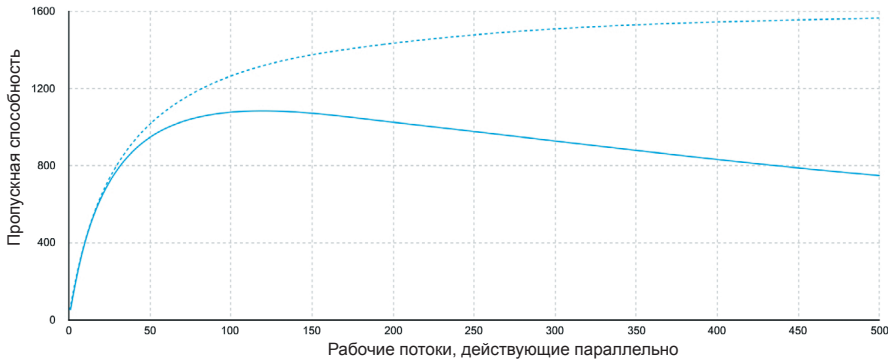


Рис. 6.7. Зависимость пропускной способности от степени распараллеливания. Сравнение закона Амдала (пунктирная линия) и универсального закона масштабирования (сплошная линия)

Из этого графика видно, что существует точка перегиба, начиная с которой пропускная способность падает. Кроме того, для лучшего представления масштабируемости реальной системы на рис. 6.7 показаны графики масштабируемости систем на основе закона Амдала и универсального закона масштабирования. Кривая ухудшения времени отклика также изменила свою форму. На рис. 6.8 показано, как изменяется задержка в зависимости от степени распараллеливания.

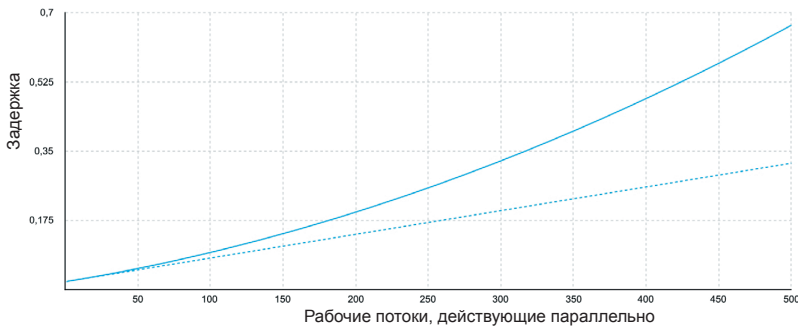


Рис. 6.8. Зависимость задержки от степени распараллеливания. Сравнение закона Амдала (пунктирная линия) и универсального закона масштабирования (сплошная линия)

Аналогично для демонстрации отличий вместе с кривой изменения задержки согласно универсальному закону масштабирования на рис. 6.8 приводится кривая изменения задержки согласно закону Амдала. Как видим, система действует иначе при наличии общих точек, для доступа к которым может требоваться дополнительная синхронизация. Схематический пример такой системы показан на рис. 6.9.

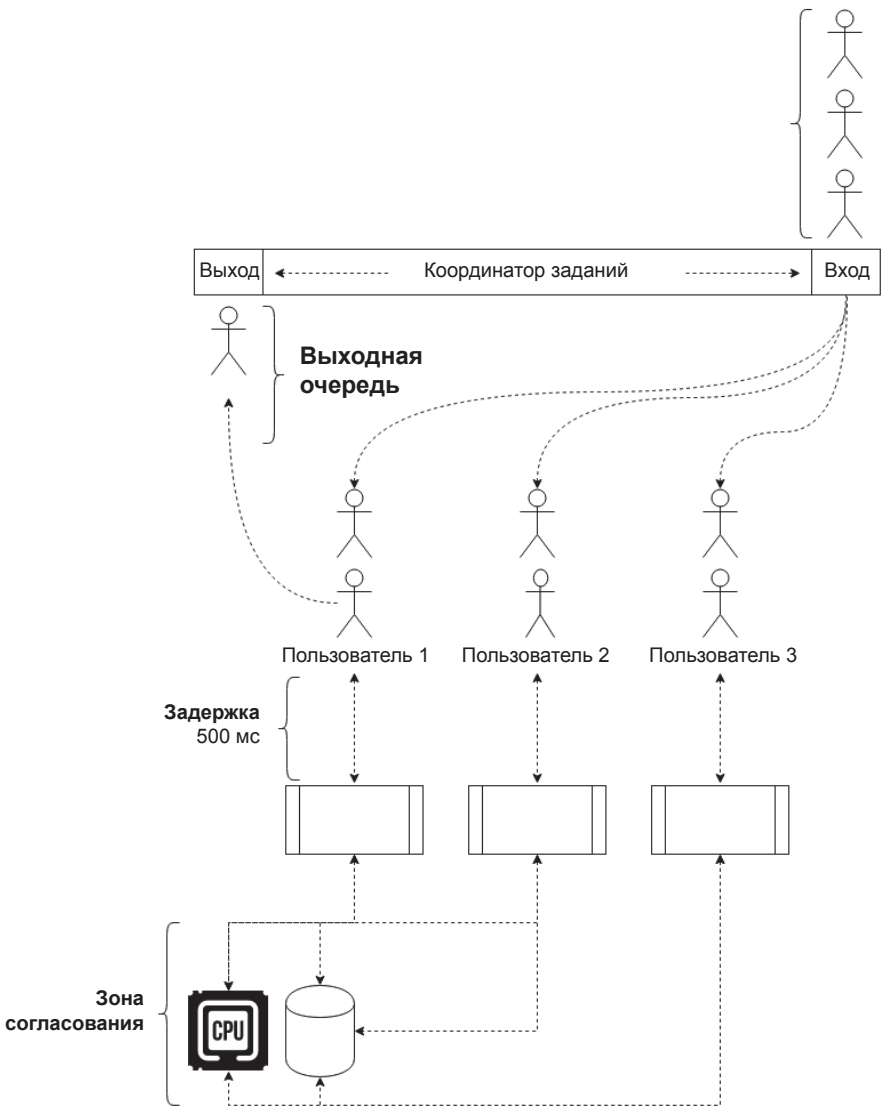


Рис. 6.9. Пример системы с параллельной обработкой заданий, соответствующей универсальному закону масштабирования

Как видите, поведение системы может быть более сложным, чем представляется с позиции закона Литтла. Существует множество скрытых ловушек, напрямую влияющих на масштабируемость системы.

В завершение этих трех разделов подчеркнем, что знание перечисленных законов играет важную роль в моделировании масштабируемых систем и планировании их емкости. Эти законы применимы и к сложным высоконагруженным распре-

деленным системам, и к многопроцессорным узлам с веб-приложением, основанным на Spring Framework. Кроме того, знание аспектов, влияющих на масштабируемость системы, позволяет правильно спроектировать систему и избежать ловушек, таких как несогласованность и конкуренция. Это знание поможет также правильно оценить модули WebFlux и Web MVC и спрогнозировать, какой подход к масштабированию поможет добиться лучших результатов.

Анализ и сравнение

Мы уже знаем, насколько важно понимать особенности поведения фреймворков, их архитектуры и модели использования ресурсов. Кроме того, очень важно выбрать фреймворк, подходящий для решения конкретной задачи. В следующих подразделах сравним Web MVC и WebFlux с разных точек зрения и узнаем, для каких областей лучше всего подходит каждый из них.

Модели обработки в WebFlux и Web MVC

Прежде всего, чтобы понять влияние различных моделей обработки на пропускную способность и задержку системы, мы должны вспомнить, как происходит обработка входящих запросов в Web MVC и WebFlux.

Как отмечалось выше, Web MVC построен с использованием блокирующих операций ввода/вывода. Это означает, что поток выполнения, обрабатывающий каждый входящий запрос, может быть заблокирован операцией чтения тела запроса из канала ввода/вывода, как показано на рис. 6.10.

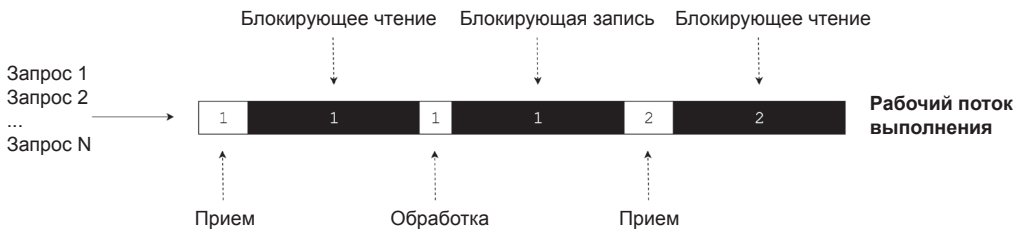


Рис. 6.10. Блокирующая обработка запроса и ответа

В примере, изображенном на рис. 6.10, все запросы ставятся в очередь и обрабатываются последовательно одним потоком выполнения Thread. Черные прямоугольники соответствуют периодам выполнения блокирующих операций ввода/вывода. Также отметим, что фактическое время обработки (белые прямоугольники) существенно меньше времени, потраченного на блокирующие операции. Из этой простой диаграммы можно сделать вывод, что поток выполнения работает неэффективно и время, расходуемое на ожидание, вполне можно было бы потратить на прием и обработку других запросов из очереди.

Модуль *WebFlux*, напротив, построен на основе неблокирующего API. То есть рабочему потоку выполнения не требуется выполнять никаких блокирующих операций ввода/вывода. Этот эффективный подход к обработке запросов изображен на рис. 6.11.

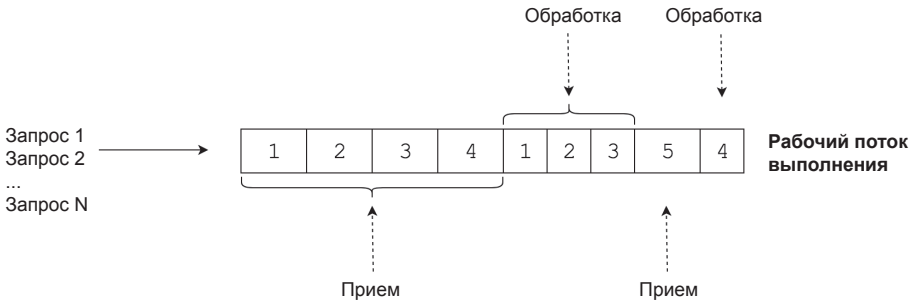


Рис. 6.11. Асинхронная и неблокирующая обработка запросов

Как показано на рис. 6.11, мы имеем случай, идентичный предыдущему, с блокирующим вводом/выводом. Слева на диаграмме имеется очередь запросов, а в середине – временной график обработки. В данном случае на временном графике отсутствуют черные прямоугольники, то есть если из Сети поступило недостаточно байтов для продолжения обработки запроса, мы всегда можем переключиться на обработку другого запроса, не блокируя поток выполнения *Thread*. Сравнивая асинхронную и неблокирующую обработку запросов с блокирующей обработкой, заметим, что теперь вместо ожидания сборки тела запроса поток выполнения может потратить время более продуктивно, например чтобы принять новое соединение. Затем операционная система может уведомить, что тело запроса получено полностью, и приложение сможет продолжить его обработку без блокировки. В этом случае процессорное время расходуется более оптимально. Аналогично без блокировки выполняется отправка ответа. Единственное отличие – система должна уведомить приложение, когда все будет готово к записи следующей порции данных без блокировки.

Предыдущий пример показывает, что *WebFlux* намного эффективнее использует единственный поток выполнения, чем *Web MVC*, и поэтому может обработать большее число запросов за то же время. Однако в нашем распоряжении все еще остается возможность многопоточного выполнения, а значит, мы можем использовать всю мощь современных процессоров, создав надлежащее количество экземпляров *Thread*. Следовательно, для более быстрой обработки запросов и достижения того же уровня загрузки процессора с применением блокирующего фреймворка *Web MVC* можем запустить не один, а несколько потоков выполнения или даже создать отдельный поток для каждого соединения, как показано на рис. 6.12.



Рис. 6.12. Создание по одному потоку выполнения для каждого соединения в Web MVC

Как показано на рис. 6.12, многопоточная модель позволяет быстрее обрабатывать запросы из очереди и создает иллюзию, что система способна принимать и обрабатывать почти то же число запросов.

Однако это решение имеет свои недостатки. Как мы уже знаем из универсального закона масштабирования, когда система имеет общие ресурсы, такие как процессор или память, увеличение числа рабочих потоков выполнения может ухудшить общую производительность системы. В данном случае, когда обработка запросов выполняется слишком большим количеством потоков выполнения, имеет место снижение производительности из-за несогласованности между ними.

Влияние моделей обработки на пропускную способность и задержку

Для проверки этого утверждения попробуем провести простые нагрузочные испытания. Используем простое приложение на основе Spring Boot 2.x, применяющее модуль Web MVC или WebFlux (назовем его промежуточным ПО). Также симулируем поток ввода/вывода из промежуточного ПО, выполнив несколько сетевых запросов к сторонней службе, возвращающей признак успеха с гарантированной 200-миллисекундной средней задержкой. Схема взаимодействий изображена на рис. 6.13.

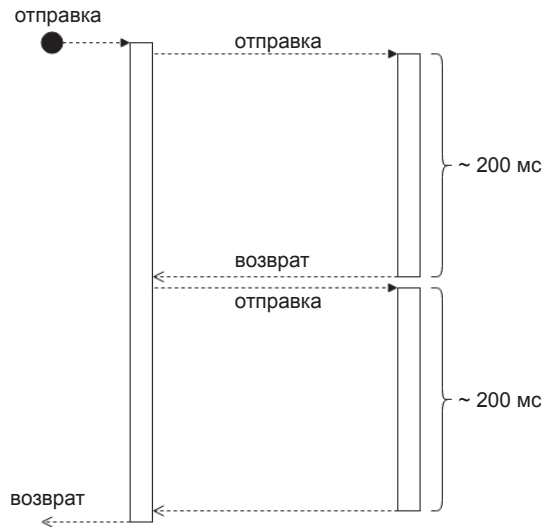


Рис. 6.13. Схема взаимодействий в нагрузочных испытаниях

Для запуска промежуточного ПО и имитации активности клиента используем инфраструктуру Microsoft Azure с ОС Ubuntu Server 16.04 на каждой машине. Для промежуточного ПО используем D12 v2 VM (четыре виртуальных процессора и 28 Гбайт ОЗУ). Для клиента используем F4 v2 VM (четыре виртуальных процессора и 8 Гбайт ОЗУ). Пользовательская активность будет постепенно увеличиваться. В начале испытаний с системой одновременно будут работать четыре пользователя, а к концу их число увеличится до 20 тыс. Это даст нам гладкую кривую изменения задержки и пропускной способности и позволит построить наглядные графики. Чтобы обеспечить соответствующую нагрузку на промежуточное ПО и собрать статистику измерений, используем современный инструмент тестирования производительности – **wrk** (<https://github.com/wg/wrk>).



Обратите внимание, что целью нашего испытания является выявление *тенденции*, а не определение *стабильности* системы, а также оценка качества реализации фреймворка WebFlux. Измерения, демонстрируемые далее, показывают превосходство неблокирующих и асинхронных взаимодействий в WebFlux над блокирующей реализацией в Web MVC, основанной на использовании множества потоков выполнения.

Ниже приводится код промежуточного ПО с Web MVC, использованный в испытаниях.

```
@RestController                                     // (1)
@SpringBootApplication                                 //
public class BlockingDemoApplication                 //
    implements InitializingBean {                    //
    ...                                              // (1.1)
```



```

@GetMapping("/") // (2)
public void get() {                                     //
    restTemplate.getForObject(someUri, String.class);   // (2.1)
    restTemplate.getForObject(someUri, String.class);   // (2.2)
}                                                       //
...                                                    //
}                                                       //

```

Пояснения к коду.

1. Объявление класса с аннотацией `@SpringBootApplication`. Вместе с тем этот класс является также контроллером, о чем сообщает аннотация `@RestController`. Чтобы максимально упростить пример, мы опустили код инициализации и объявление полей (1.1).
2. Объявление метода `get` с аннотацией `@GetMapping`. Чтобы уменьшить избыточный сетевой трафик и все внимание сосредоточить только на производительности фреймворка, мы ничего не возвращаем в теле ответа. Согласно диаграмме на рис. 6.13 мы посылаем два HTTP-запроса удаленному серверу, как показано в строках (2.1) и (2.2).

Судя по предыдущему примеру и диаграмме на рис. 6.13, среднее время отклика промежуточного ПО должно составлять примерно 400 мс.

Обратите внимание, что для этого испытания мы задействуем веб-сервер Tomcat, который по умолчанию используется с Web MVC. Кроме того, чтобы увидеть, как меняется производительность Web MVC, создадим несколько экземпляров Thread, по числу одновременно обслуживаемых пользователей. Настройки Tomcat иллюстрирует следующий сценарий sh.

```

java -Xss512K -Xmx24G -Xms24G
-Dserver.tomcat.prestartmin-spare-threads=true
-Dserver.tomcat.prestart-min-spare-threads=true
-Dserver.tomcat.max-threads=$1
-Dserver.tomcat.min-spare-threads=$1
-Dserver.tomcat.max-connections=100000
-Dserver.tomcat.accept-count=100000
-jar ...

```

Как демонстрирует этот сценарий, значения параметров `max-threads` и `min-spare-threads` выбираются динамически и определяются числом одновременно обслуживаемых пользователей.



Предыдущая конфигурация не предназначена для использования в промышленном окружении. Ее цель – показать недостатки модели Spring Web MVC, когда для обслуживания каждого пользователя запускается отдельный поток выполнения.

Запустив испытываемое программное обеспечение, мы получили кривую, изображенную на рис. 6.14.



Рис. 6.14. Результаты измерения пропускной способности Web MVC

Диаграмма на рис. 6.14 наглядно показывает, что начиная с некоторого момента пропускная способность неуклонно снижается. Это означает существенное влияние конкуренции и несогласованности, имеющих место в нашем приложении.

Для сравнения производительности мы подготовили идентичное приложение на основе WebFlux, представленное ниже.

```
@RestController
@SpringBootApplication
public class ReactiveDemoApplication
    implements InitializingBean {
    ...
    @GetMapping("/")
    public Mono<Void> get() { // (1)
        return //
            WebClient //
                .get() // (2)
                .uri(someUri) //
                .retrieve() //
                .bodyToMono(DataBuffer.class) //
                .doOnNext(DataBufferUtils::release) //
            .then() // (3)
            WebClient //
                .get() // (4)
                .uri(someUri) //
                .retrieve() //
                .bodyToMono(DataBuffer.class) //
                .doOnNext(DataBufferUtils::release) //
                .then() //
    }
}
```

```
        .then(); // (5)
    }
    ...
}
```

На этот раз активно используем возможности Spring WebFlux и Project Reactor, чтобы обеспечить асинхронную и неблокирующую обработку запросов и ответов. Так же как в случае с Web MVC, в строке (1) возвращаем результат `Void`, но теперь завернутый в реактивный тип `Mono`. Производим удаленный вызов с помощью `WebClient API` и затем (3) последовательно выполняем точно такой же второй удаленный вызов (4). Далее, не дожидаясь результатов, возвращаем `Mono<Void>`, который уведомит подписчика, когда оба запроса будут выполнены.



Обратите внимание, что с использованием библиотеки Reactor можно улучшить время выполнения, даже когда два запроса посылаются последовательно. Поскольку обе операции выполняются асинхронно и не блокируют процесс, нет необходимости создавать дополнительные экземпляры `Thread`. Но для корректного сравнения мы сохранили поведение системы, описываемое диаграммой на рис. 6.13, и выполняем два запроса последовательно, поэтому задержка в этой версии приложения тоже должна составлять в среднем примерно 400 мс.

Запустив испытываемое программное обеспечение на основе WebFlux, мы получили кривую, изображенную на рис. 6.15.

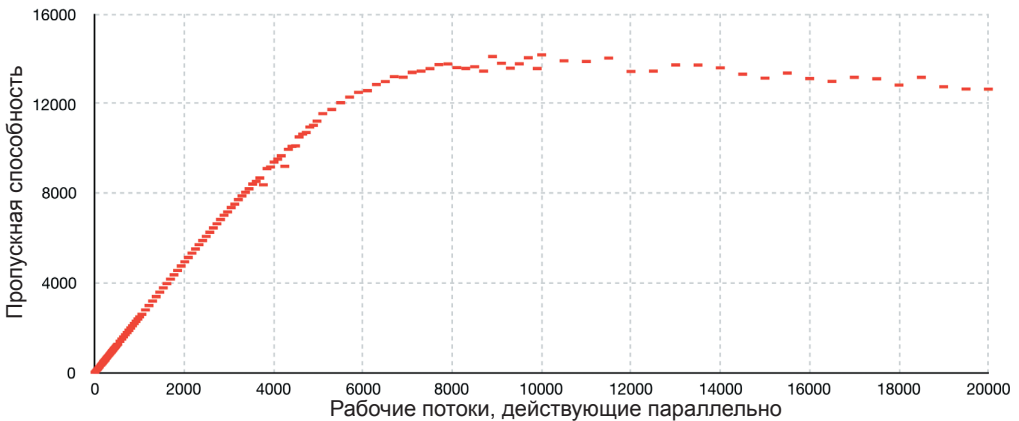


Рис. 6.15. Результаты измерения пропускной способности WebFlux

Как видите, кривая пропускной способности WebFlux похожа на кривую для WebMVC.

Однако, чтобы сравнение получилось более наглядным, совместим эти две кривые на одном графике, как показано на рис. 6.16.

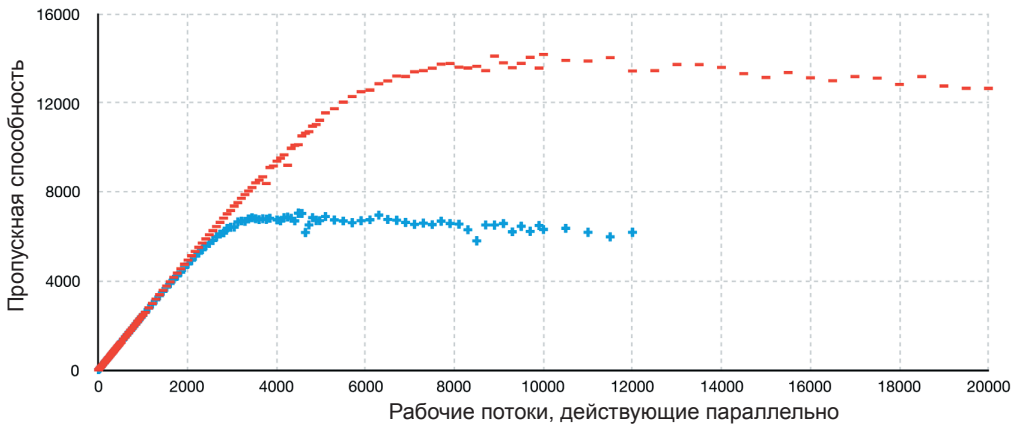


Рисунок 6.16. Сравнение пропускной способности WebFlux и Web MVC

На рис. 6.16 кривая из символов + (плюс) представляет график изменения пропускной способности Web MVC, а кривая из символов – (минус) – график изменения пропускной способности WebFlux. В данном случае чем выше, тем лучше. Как видите, пропускная способность WebFlux почти в два раза выше.

Следует также отметить, что для Web MVC отсутствуют измерения для числа пользователей больше 12 тыс. Проблема в том, что пул в Tomcat с большим числом потоков выполнения не уместается в заданном объеме памяти (28 Гбайт). Поэтому всякий раз, когда Tomcat пытался создать больше 12 тыс. экземпляров Thread, ядро Linux останавливало процесс. Это лишний раз подчеркивает, что модель *один поток для каждого соединения* не подходит для случаев, когда требуется одновременно обслужить более 10 тыс. пользователей.



Предыдущий пример сравнивает модель *один поток для каждого соединения* с моделью асинхронной и неблокирующей обработки. В первом случае единственный способ обработать запросы, избежав существенного влияния на задержку, – создать отдельный поток выполнения для каждого пользователя. То есть мы минимизируем время ожидания пользователя в очереди, пока освободится поток выполнения. В конфигурации с использованием WebFlux, напротив, нет необходимости выделять самостоятельный поток выполнения для каждого пользователя, потому что операции ввода/вывода выполняются в асинхронном и неблокирующем режиме. Кроме того, на практике сервер Tomcat имеет ограниченный размер пула потоков.

Тем не менее обе кривые показывают схожую тенденцию/обе имеют критические точки, после которых пропускная способность начинает уменьшаться. Это можно объяснить тем, что многие системы имеют свои ограничения числа открытых соединений с клиентами. Кроме того, сравнение может быть не совсем корректным,

потому что мы используем разные реализации HTTP-клиентов с разными настройками. Например, `RestTemplate` по умолчанию использует стратегию, согласно которой для каждого нового запроса создается новое HTTP-соединение, тогда как `WebClient` на основе `Netty` по умолчанию использует пул соединений. В последнем случае соединения могут использоваться повторно. Но даже если первую систему настроить на повторное использование открытых соединений, сравнение все равно будет не совсем корректным.

Поэтому, чтобы увеличить корректность сравнения, симитируем сетевые операции, отмеряя 400-миллисекундные задержки программно. Для этого в обоих случаях используем такой код.

```
Mono.empty()  
    .delaySubscription(Duration.ofMillis(200))  
    .then(Mono.empty())  
    .delaySubscription(Duration.ofMillis(200)))  
    .then()
```

Для случая с `WebFlux` будет возвращаться тип `Mono<Void>`, а для случая с `Web MVC` выполнение будет завершаться вызовом `.block()`, то есть поток выполнения будет блокироваться на указанную величину задержки. Чтобы получить идентичное поведение, формирующее задержку, используем один и тот же код.

Также для обоих случаев применяем похожую облачную конфигурацию. Для промежуточного ПО будет использоваться экземпляр `E4S V3 VM` (четыре виртуальных процессора и 32 Гбайт ОЗУ) и для клиента – `B4MS VM` (четыре виртуальных процессора и 16 Гбайт ОЗУ).

Опробовав это решение, мы получили результаты, изображенные на рис. 6.17.

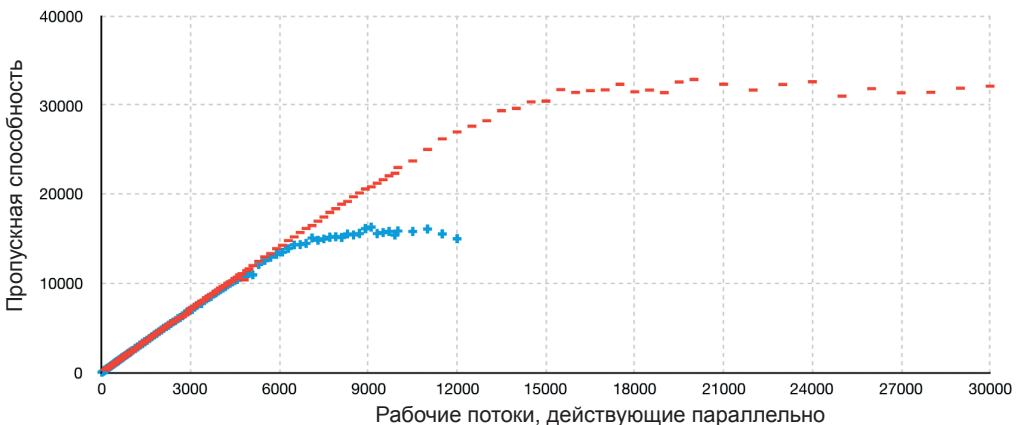


Рис. 6.17. Сравнение пропускной способности `WebFlux` и `Web MVC` без дополнительного ввода/вывода

На рис. 6.17 кривая из символов + (плюс) представляет график изменения пропускной способности Web MVC, а кривая из символов – (минус) – график изменения пропускной способности WebFlux. Как видите, пропускная способность в обоих случаях оказалась выше, чем в ситуации, когда использовались реальные сетевые операции. Это означает, что наличие пула соединений в приложении и политика управления соединениями в операционной системе оказывают существенное влияние на производительность.

И тем не менее WebFlux все равно показывает в два раза большую пропускную способность, чем Web MVC, что еще раз подтверждает наше предположение о неэффективности модели *один поток для каждого соединения*. WebFlux по-прежнему действует в точном соответствии с законом Амдала. Однако важно помнить, что кроме ограничений, накладываемых приложением, есть ограничения, накладываемые системой, которые могут влиять на интерпретацию конечных результатов.

Мы можем также сравнить модули WebFlux и Web MVC по величине задержки и потреблению времени процессора (соответствующие графики показаны на рис. 6.18 и 6.19 соответственно).

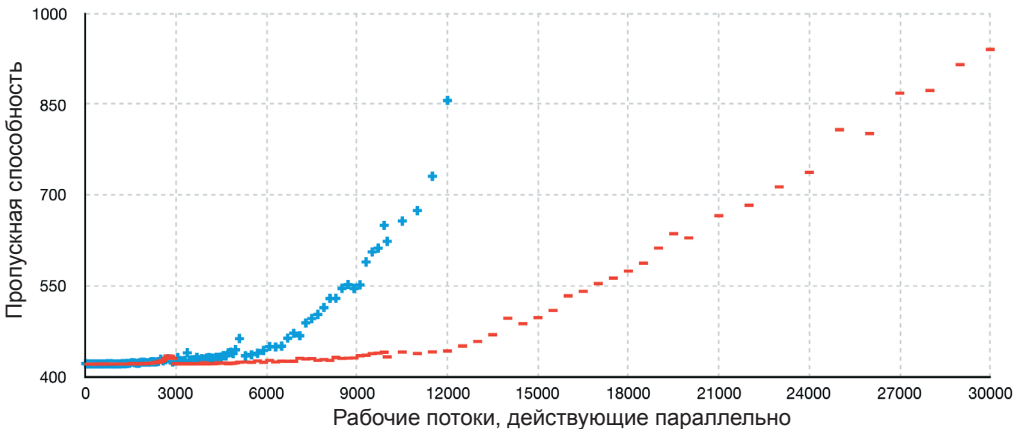


Рис. 6.18. Сравнение задержки WebFlux и Web MVC баз дополнительного ввода/вывода

На рис. 6.18 кривая из символов + (плюс) представляет график изменения задержки в Web MVC, а кривая из символов – (минус) – график изменения задержки в WebFlux. В данном случае чем ниже значение, тем лучше. Диаграмма на рис. 6.18 показывает быстрое увеличение времени задержки для Web MVC. Для 12 тыс. одновременно обслуживаемых пользователей WebFlux показывает примерно в 2,1 раза лучшее время задержки.

В отношении использования процессорного времени мы получили следующую тенденцию (см. рис. 6.19).

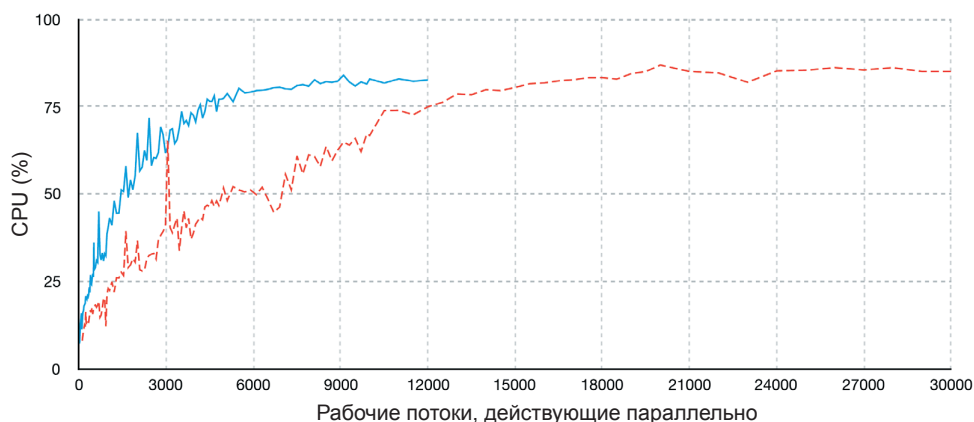


Рис. 6.19. Сравнение использования процессорного времени WebFlux и Web MVC баз дополнительного ввода/вывода

На рис. 6.19 сплошная линия представляет Web MVC, пунктирная – WebFlux. И снова в этом случае чем ниже результат, тем лучше. Глядя на рис. 6.19, можно заключить, что WebFlux действует намного эффективнее в терминах пропускной способности, задержки и потребления процессорного времени. Разницу в потреблении процессорного времени можно объяснить избыточной работой, связанной с переключением контекста между разными экземплярами Thread.

Проблемы модели обработки в WebFlux

WebFlux значительно отличается от Web MVC. Благодаря отсутствию блокирующего ввода/вывода для обработки всех запросов достаточно небольшого количества экземпляров Thread. Для одновременной обработки событий не требуется потоков выполнения больше, чем число процессоров/ядер в системе.



Это объясняется тем, что WebFlux основан на Netty, где по умолчанию число экземпляров Thread выбирается равным значению `Runtime.getRuntime().availableProcessors()`, умноженному на два.

Использование неблокирующих операций позволяет обрабатывать результаты асинхронно (см. рис. 6.11), обеспечивает лучшую масштабируемость и более эффективное использование процессорного времени, позволяет тратить такты процессора на фактическую обработку и снижать затраты на переключение контекста, тем не менее асинхронная, неблокирующая модель имеет свои недостатки. Прежде всего важно понимать, что вычислительные задания должны планироваться для выполнения в отдельных экземплярах Thread или ThreadPool. Эта проблема не относится к модели *один поток для каждого соединения*, где пул имеет большое количество рабочих потоков, потому что в этом случае каждому соединению соответствует свой выделенный рабочий поток. Обычно большинство разработчиков, имеющих богатый опыт использования такой модели, забывают

об этом и выполняют вычислительные задания в главном потоке. Такая забывчивость обходится дорого и может повлиять на общую производительность. В этом случае главный поток окажется занят обработкой и не будет успевать принимать и обрабатывать новые соединения.

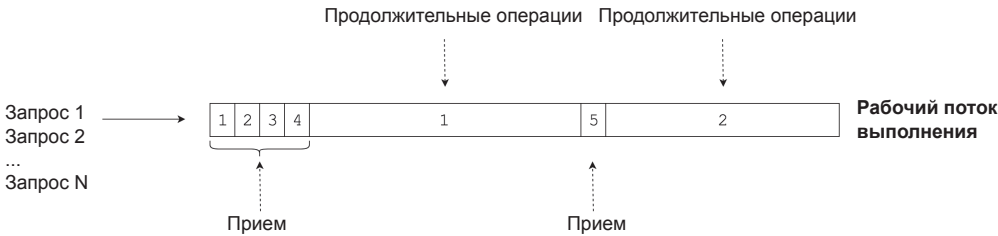


Рис. 6.20. Выполнение вычислительных заданий в однопроцессорном окружении

Как показано на рис. 6.20, даже притом что весь временной график состоит только из белых прямоугольников (что означает отсутствие блокирующего ввода/вывода), интенсивные вычисления крадут время, которое можно было бы потратить на обработку других запросов.

Чтобы решить эту проблему, следует делегировать выполнение продолжительных операций отдельному пулу процессоров или, если узел имеет единственный процессор, другим узлам. Например, можно организовать эффективный цикл обработки событий (https://ru.wikipedia.org/wiki/Цикл_событий), когда один экземпляр Thread принимает соединения и делегирует фактическую их обработку отдельному пулу потоков/узлов, как показано на рис. 6.21.

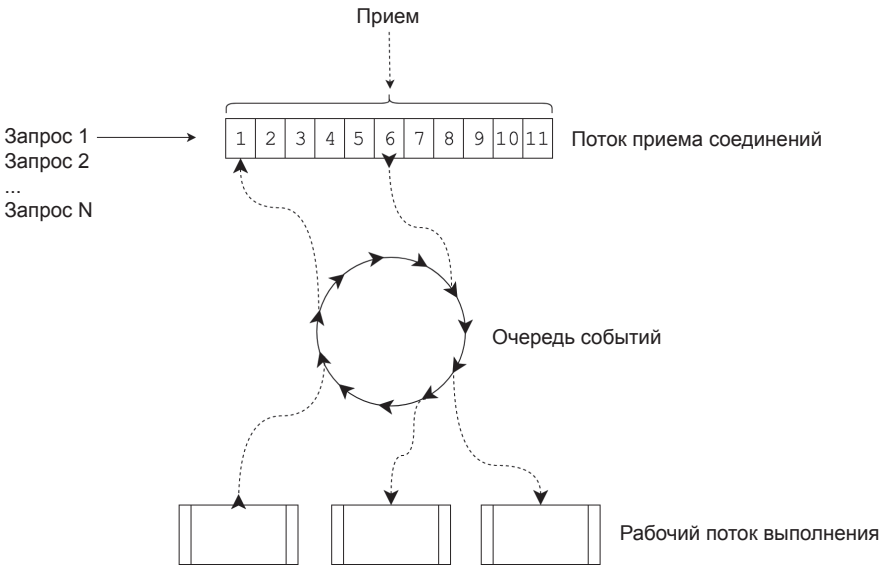


Рис. 6.21. Netty-подобная неблокирующая серверная архитектура

Другая распространенная ошибка, связанная с асинхронным и неблокирующим программированием, – использование блокирующих операций. Одна из самых нетривиальных операций в разработке веб-приложений – генерирование уникального UUID.

```
UUID requestUniqueId = java.util.UUID.randomUUID();
```

Проблема здесь в том, что `#randomUUID()` использует `SecureRandom`. Обычно криптостойкие генераторы случайных чисел используют внешний источник энтропии по отношению к приложению. Это может быть аппаратный генератор случайных чисел, но чаще это механизм накопленной случайности, который используется операционной системой при нормальной работе.



В данном контексте под случайностью подразумеваются такие события, как движения мышью, изменения напряжения в сети электропитания и другие случайные события, которые могут выбираться системой в процессе работы.

Проблема в том, что источники энтропии имеют ограниченную емкость. Если она будет исчерпана, системный вызов чтения энтропии заблокируется до появления новых случайных событий. Кроме того, наличие большого потока выполнения существенно влияет на скорость генерирования идентификаторов UUID. Причины легко понять, заглянув в реализацию метода `SecureRandom#nextBytes(byte[] bytes)`, который генерирует случайные числа для `UUID.randomUUID()`.

```
synchronized public void nextBytes(byte[] bytes) {  
    secureRandomSpi.engineNextBytes(bytes);  
}
```

Как видите, `#nextBytes` синхронизируется, что влечет за собой значительную потерю производительности при попытке обращения к нему из нескольких потоков выполнения.



Узнать больше о производительности `SecureRandom` можно на сайте Stack Overflow: <https://stackoverflow.com/questions/137212/how-to-solve-slow-java-securerandom>.

Как мы уже знаем, *WebFlux* использует несколько потоков выполнения для асинхронной и неблокирующей обработки большого числа запросов. Поэтому нужно быть осторожными, используя методы, которые на первый взгляд не выглядят операциями ввода/вывода, но в действительности скрывают некоторые особенные взаимодействия с операционной системой. Без должного внимания к таким методам можно сильно ухудшить общую производительность системы. Поэтому при работе с *WebFlux* важно использовать только неблокирующие операции. Однако такое требование накладывает множество ограничений на применение приемов реактивной разработки. Например, вся библиотека *Java Development Kit* проектировалась исходя из предположения использования императивных, син-

хронных взаимодействий между компонентами экосистемы Java. Поэтому многие блокирующие операции не имеют неблокирующих, асинхронных аналогов, что усложняет разработку по-настоящему реактивных систем. WebFlux дает нам более высокую пропускную способность и низкую задержку, но взамен требует с большим вниманием относиться ко всем операциям и библиотекам, с которыми мы работаем.

Кроме того, когда сложные вычисления составляют основу нашей службы, несложная многопоточная модель оказывается предпочтительнее, чем модель неблокирующей и асинхронной обработки. И даже если все операции ввода/вывода являются блокирующими, у нас остается не так много преимуществ перед неблокирующим вводом/выводом. Более того, сложность неблокирующих и асинхронных алгоритмов обработки событий может оказаться избыточной, и более простая многопоточная модель Web MVC может оказаться эффективнее WebFlux.

Однако, когда отсутствуют существенные ограничения или необычные варианты использования и приложение выполняет большое количество операций ввода/вывода, неблокирующий и асинхронный WebFlux может засиять новыми гранями.

Потребление памяти разными моделями обработки

Еще одна важная характеристика фреймворков, заслуживающая сравнения, – потребление памяти. Из обсуждения модели *один поток для каждого соединения* в главе 1 «Причины выбора Spring» мы знаем, что вместо выделения памяти для каждого крошечного объекта, представляющего событие, предпочтительнее выделить один большой экземпляр Thread для каждого соединения. Первое, что мы должны иметь в виду, – экземпляр Thread резервирует некоторое пространство на стеке. Фактический размер стека зависит от операционной системы и настроек JVM. По умолчанию на большинстве 64-разрядных серверов для виртуальной машины Java выделяется стек размером 1 Мбайт.



Под событием подразумевается сигнал об изменении состояния системы, таком как открытие соединения или получение новой порции данных.

Приложения, действующие под высокой нагрузкой и реализованные по этому принципу, будут потреблять большие объемы памяти. Самое плохое, что может случиться, – это напрасное выделение 1 Мбайта стека для хранения тела запроса и ответа. Если размер выделенного пула ограничить, снизится пропускная способность и увеличится средняя задержка. Поэтому, используя Web MVC, необходимо искать хороший баланс между потребляемой памятью и пропускной способностью. При использовании WebFlux, напротив, как мы узнали в предыдущем разделе, можно задействовать фиксированное число экземпляров Thread, обрабатывать намного больше запросов и использовать предсказуемый объем па-

мяти. Чтобы получить более полное представление о предыдущих испытаниях, сравним потребление памяти в них (см. рис. 6.22).

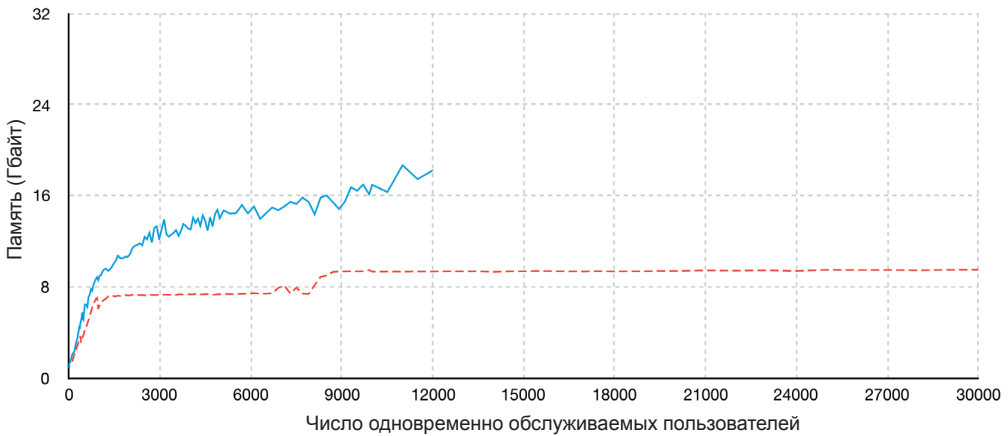


Рис. 6.22. Сравнение потребления памяти в решениях на основе WebFlux и Web MVC

На рис. 6.22 сплошная линия представляет Web MVC, пунктирная – WebFlux. В этом случае чем ниже результат, тем лучше. Также следует отметить, что для обоих приложений дополнительно были настроены параметры JVM: *Xms26GB* и *Xmx26GB*. То есть оба приложения имели в своем распоряжении один и тот же объем памяти. Однако в приложении на основе Web MVC потребление памяти увеличивалось пропорционально числу одновременно обслуживаемых запросов. Как упоминалось в начале этого раздела, обычно для каждого экземпляра Thread выделяется стек размером 1 Мбайт. Для данного примера выбран размер стека потока выполнения *-Xss512K*, поэтому каждый поток получал дополнительно ~512 Кбайт памяти. То есть модель *один поток для каждого соединения* неэффективно использовала память.

В приложении на основе WebFlux, напротив, наблюдается стабильность потребления памяти независимо от числа одновременно обслуживаемых запросов. Это означает, что WebFlux оптимальнее использует память. Иначе говоря, с WebFlux мы можем использовать более дешевые серверы.

Чтобы убедиться в правильности этого вывода, проведем небольшой эксперимент, касающийся предсказуемости потребления памяти, и посмотрим, как это поможет в непредсказуемых ситуациях. Проанализируем, сколько денег мы потратили бы на облачную инфраструктуру с Web MVC и WebFlux. Для оценки верхнего предела системы проведем стресс-тестирование и проверим, сколько запросов сможет обработать система. Запустим веб-приложение на экземпляре Amazon EC2 *t2.small* с одним виртуальным процессором и 2 Гбайт ОЗУ. В роли операционной системы используем Amazon Linux с установленной библиотекой JDK 1.8.0_144 и VM 25.144-b01. Для первого цикла измерений используем Spring

Boot 2.0.x и Web MVC с Tomcat. Для имитации сетевых вызовов и другой активности, связанной с вводом/выводом, обычной для современных систем, используем следующий простой код:

```
@RestController
@SpringBootApplication
public class BlockingDemoApplication {
    ...
    @GetMapping("/endpoint")
    public String get() throws InterruptedException {
        Thread.sleep(1000);
        return "Hello";
    }
}
```

Запускать приложение будем следующей командой:

```
java -Xmx2g
    -Xms1g
    -Dserver.tomcat.max-threads=20000
    -Dserver.tomcat.max-connections=20000
    -Dserver.tomcat.accept-count=20000
    -jar blocking-demo-0.0.1-SNAPSHOT.jar
```

Используя предыдущую конфигурацию, выясним, сможет ли наша система без сбоев обслуживать до 20 тыс. пользователей одновременно. Проведя эти нагрузочные испытания, мы получили следующие результаты.

Число одновременно обрабатываемых запросов	Средняя задержка (миллисекунд)
100	1271
1000	1429
10 000	OutOfMemoryError

Эти результаты могут немного меняться от случая к случаю, но в среднем будут идентичны. Как видите, 2 Гбайт памяти недостаточно для запуска 10 тыс. потоков выполнения, каждый из которых обслуживает свое соединение. Разумеется, поэкспериментировав с настройками JVM и Tomcat, можно немного улучшить результаты, но это не решает проблемы чрезмерного расходования памяти. Сохранив тот же сервер приложений и просто переключившись на WebFlux над Servlet 3.1, можно добиться более существенного улучшения. Вот как выглядит новое веб-приложение.

```
@RestController
@SpringBootApplication
public class TomcatNonBlockingDemoApplication {
```

```
...
@GetMapping("/endpoint")
public Mono<String> get() {
    return Mono.just("Hello")
        .delaySubscription(Duration.ofSeconds(1));
}
```

В этом случае моделируется неблокирующий и асинхронный ввод/вывод, чего легко добиться с помощью Reactor 3 API.



Обратите внимание, что в качестве серверного движка для WebFlux используется Reactor-Netty. Поэтому, чтобы переключиться на использование веб-сервера Tomcat, нужно исключить зависимость `spring-boot-starter-reactor-netty` из настроек WebFlux и добавить зависимость от `spring-boot-starter-tomcat`.

Запуск нового стека осуществлялся следующей командой:

```
java -Xmx2g
     -Xms1g
     -Dserver.tomcat.accept-count=20000
     -jar non-blocking-demo-tomcat-0.0.1-SNAPSHOT.jar
```

Мы выбрали точно такое же распределение памяти для нашего приложения на Java, но на этот раз использовали размер пула потоков, по умолчанию равный 200 потокам. Проведя те же испытания, мы получили следующие результаты.

Число одновременно обрабатываемых запросов	Средняя задержка (миллисекунд)
100	1203
1000	1407
10 000	9661

Как видите, в этом случае наше приложение показывает намного лучшие результаты. Однако они все еще далеки от идеала, поскольку в периоды высокой нагрузки пользователи будут вынуждены ждать ответа слишком долго. Давайте посмотрим, как изменится пропускная способность и задержка, если использовать по-настоящему реактивный сервер Reactor-Netty.

Поскольку код приложения и команда не изменились, мы не будем приводить их, а сразу перейдем к результатам испытаний.

Число одновременно обрабатываемых запросов	Средняя задержка (миллисекунд)
1000	1307
10 000	2699
20 000	6310

Как видите, результаты получились еще лучше. Прежде всего в результатах для Netty мы указали минимальное число одновременно обрабатываемых запросов, равное 1000, а максимальное – равное 20 000. Это наглядно показывает, что сервер Netty позволяет добиться в два раза более высокой производительности, чем Tomcat. Только одно это сравнение показывает, что решения на основе WebFlux могут снизить стоимость инфраструктуры, потому что приложения потребляют намного меньше ресурсов и могут действовать на недорогих серверах.

Другая выгода, которую дает модуль WebFlux, – возможность быстрее обрабатывать тело входящего запроса с меньшим потреблением памяти. Эта выгода становится особенно очевидной, когда тело запроса представлено коллекцией элементов и есть возможность обрабатывать их по отдельности (см. рис. 6.23).

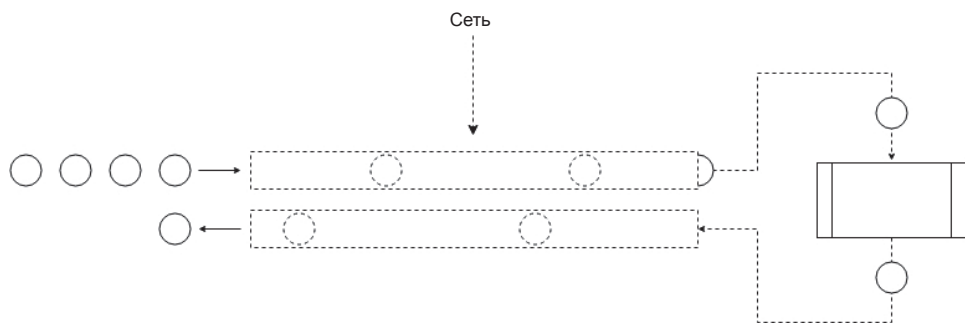


Рис. 6.23. WebFlux обрабатывает большой массив небольшими порциями



Узнать больше о кодировании/декодировании сообщений можно по ссылке <https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-codecs>.

Как можно судить по диаграмме на рис. 6.23, чтобы система могла начать обработку тела запроса, достаточно иметь небольшую его часть. Аналогично можно выполнять передачу тела ответа клиенту. Нет необходимости ждать, когда тело ответа будет сконструировано целиком. Его элементы можно отправлять по мере готовности. Следующий код демонстрирует, как это реализуется с помощью WebFlux.

```
@RestController
@RequestMapping("/api/json")
class BigJSONProcessorController {
```

```

@GetMapping(
    value = "/process-json",
    produces = MediaType.APPLICATION_STREAM_JSON_VALUE
)
public Flux<ProcessedItem> processOneByOne(Flux<Item> bodyFlux) {
    return bodyFlux
        .map(item -> processItem(item))
        .filter(processedItem -> filterItem(processedItem));
}

```

Как можно заметить во фрагменте кода, эта удивительная возможность доступна без внедрения во внутренние компоненты Spring WebFlux, через общедоступный API. Кроме того, использование такой модели обработки позволяет начать возврат ответа намного быстрее, поскольку время между отправкой клиентом первого элемента и получением ответа вычисляется по следующей формуле:

$$R = R_{net} + R_{processing} + R_{net}.$$



Имейте в виду, что методы потоковой обработки данных не позволяют спрогнозировать размер тела ответа, что можно считать недостатком.

Модуль Web MVC, напротив, ждет, когда весь запрос будет записан в память, и только после этого обрабатывает его тело (см. рис. 6.24).

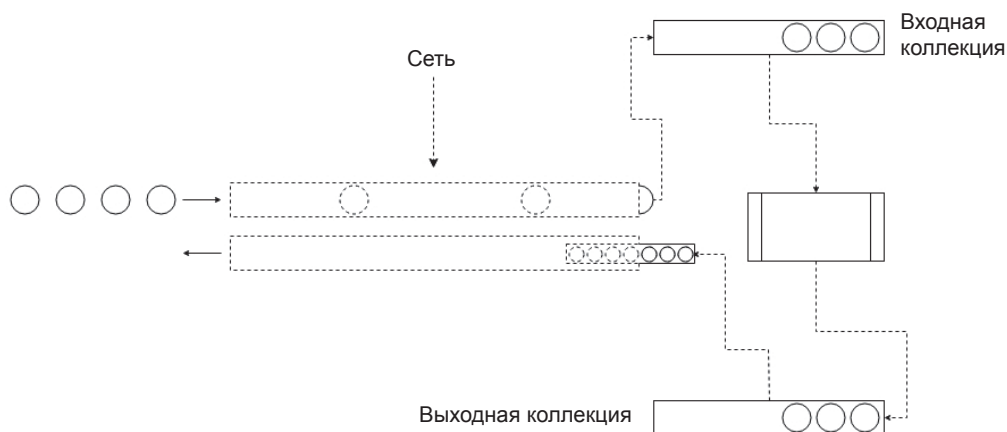


Рис. 6.24. Web MVC обрабатывает большой массив целиком

В этом случае невозможно обрабатывать данные реактивно, как в WebFlux, потому что типичное объявление `@Controller` выглядит так, как показано ниже.

```

@RestController
@RequestMapping("/api/json")
class BigJSONProcessorController {

```

```
@GetMapping("/process-json")
public List<ProcessedItem> processOneByOne(
    List<Item> bodyList
) {
    return bodyList
        .stream()
        .map(item -> processItem(item))
        .filter(processedItem -> filterItem(processedItem))
        .collect(toList());
}
```

Здесь объявление метода явно требует, чтобы полное тело запроса было преобразовано в коллекцию конкретных элементов. С математической точки зрения среднее время обработки складывается из следующих компонентов:

$$R = N \times (R_{\text{net}} + R_{\text{processing}}) + R_{\text{net}}.$$

Опять же, чтобы вернуть первый результат пользователю, требуется обработать тело запроса целиком и собрать результаты в коллекцию. Только после этого система сможет послать ответ клиенту. По этой причине WebFlux использует намного меньше памяти, чем Web MVC. WebFlux может вернуть первый элемент ответа намного раньше, чем Web MVC, и способен обрабатывать бесконечные потоки данных.

Влияние модели обработки на удобство

Чтобы сравнение Web MVC и WebFlux было полным, мы должны сопоставить не только количественные, но и качественные показатели. Одним из самых распространенных показателей качественной оценки является кривая обучения. Web MVC – хорошо известный фреймворк. Он уже больше десяти лет активно используется в сфере разработки корпоративных приложений, и опирается на простую императивную парадигму программирования. Для бизнеса это означает, что для разработки нового проекта на основе Spring 5 и Web MVC нам проще будет найти опытных разработчиков и дешевле обучить новых. Ситуация с WebFlux существенно отличается. Прежде всего WebFlux считается новой технологией, которая еще недостаточно зарекомендовала себя и теоретически может содержать немало ошибок и уязвимостей. Асинхронная и неблокирующая парадигма программирования также может стать камнем преткновения, потому что асинхронный и неблокирующий код сложнее в отладке, как показывает опыт компании Netflix, связанный с переносом Zuul на новую модель программирования.



Асинхронное программирование базируется на обратных вызовах и управляется циклом событий. Трассировка стека цикла событий лишена всякого смысла, когда следует проследить путь обработки запроса. Это объясняется особенностями обработки событий с применением обратных вызовов, однако имеется ряд инструментов, способных помочь в этом. В отдельных случаях необработанные исключения и неправиль-

ная реакция на изменение состояния может привести к утечкам ресурсов (например, ByteBuf и дескрипторов файлов), потере ответов и многим другим неприятностям. Эти проблемы довольно сложны в отладке, потому что часто очень сложно понять, какое событие было обработано неправильно. За дополнительной информацией обращайтесь по ссылке <https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c>.

Кроме того, с точки зрения бизнеса может быть неразумно заниматься поисками высококвалифицированных инженеров с глубоким знанием асинхронного и неблокирующего программирования и особенно с опытом использования стека Netty. Обучение новых разработчиков требует много времени и средств, и нет никакой гарантии, что они кардинально освоят эту технологию. К счастью, некоторые аспекты данной проблемы можно решить с помощью библиотеки Reactor 3, упрощающей конструирование конвейеров преобразования и обработки и скрывающей от программиста особо сложные детали асинхронного программирования. К сожалению, Reactor не может решить всех проблем, и для бизнеса такие непредсказуемые вложения в обучение людей и рискованные технологии могут оказаться неоправданными.

Другой важной стороной качественного анализа является миграция существующего решения на новый реактивный стек. Несмотря на то что с самого начала разработки фреймворка команда Spring прилагает все силы, чтобы обеспечить гладкую миграцию, трудно предусмотреть все случаи. Например, тем, кто использует JSP, Apache Velocity и другие похожие технологии отображения на стороне сервера, потребуется выполнить перенос всего кода, связанного с реализацией пользовательского интерфейса. Более того, многие современные фреймворки опираются на использование ThreadLocal, что усложняет переход к асинхронной и неблокирующей модели программирования. Наряду с этим существует множество проблем, связанных с базами данных, о которых подробно рассказывается в главе 7 «Реактивный доступ к базам данных».

Практическое применение WebFlux

В предыдущих разделах мы познакомились с общим дизайном модуля WebFlux и его новыми возможностями. Провели также детальное сравнение WebFlux и Web MVC и получили представление об их достоинствах и недостатках с разных точек зрения. В заключение в этом разделе попробуем получить более полное представление о практическом применении WebFlux.

Системы на основе микросервисов

Первое очевидное применение WebFlux – системы на основе микросервисов. Наиболее отличительной чертой типичных микросервисов от монолитных при-

ложений является обилие операций ввода/вывода. Наличие ввода/вывода, особенно блокирующего, ухудшает общую отзывчивость и пропускную способность системы. Конкуренция и необходимость согласования в модели *один поток для каждого соединения* еще больше ухудшают производительность системы. То есть для систем или отдельных служб, где важную роль играют вызовы между службами, WebFlux является одним из самых эффективных решений. Примером может служить механизм управления процессом проведения платежей.

Обычно за простой операцией, такой как перевод денег между счетами, скрывается сложный механизм, включающий множество операций – поиска, проверки и фактического перевода денег. Например, посылая деньги с помощью PayPal, на первом шаге может производиться получение счетов отправителя и получателя. Далее, поскольку PayPal может переводить деньги из любой страны в любую страну, важно убедиться, что перевод не нарушит законодательства этих стран. Каждый счет может иметь свои ограничения. Наконец, у получателя может быть внутренний счет PayPal или внешний, привязанный к кредитной или дебетовой карте, поэтому в зависимости от типа счета может потребоваться выполнить дополнительный вызов внешней системы, как показано на рис. 6.25.

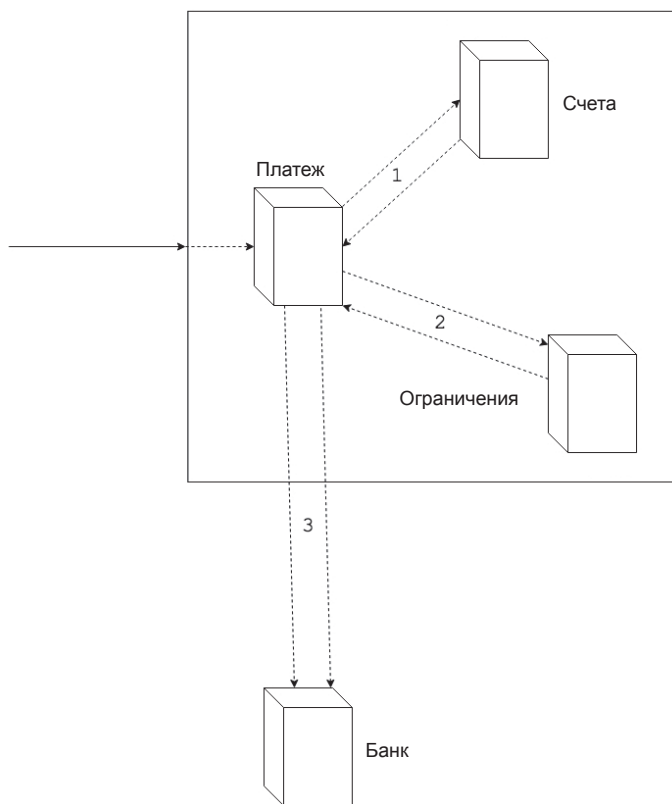


Рис. 6.25. Пример реализации процесса перевода денег через PayPal

Организовав неблокирующие и асинхронные взаимодействия в подобном сложном процессе, можно параллельно обрабатывать другие запросы и эффективно использовать ресурсы компьютера.

Системы, обслуживающие клиентов с медленными соединениями

Второе применение *WebFlux* – создание систем, предназначенных для работы с мобильными устройствами клиентов, имеющими медленное или нестабильное подключение к Сети. Чтобы понять, как *WebFlux* может пригодиться в этой ситуации, вспомним, что происходит при работе с медленным соединением. Проблема в том, что передача данных от клиента на сервер может потребовать значительного времени, точно так же как передача ответа в обратном направлении. При использовании модели *один поток для каждого соединения* высока вероятность столкнуться с аварийным поведением системы, когда число подключенных клиентов окажется слишком велико. Используя эту уязвимость, хакеры легко смогут вызвать **отказ в обслуживании**, обрушив систему.

WebFlux, в свою очередь, дает возможность принимать соединения, не блокируя рабочие потоки выполнения. Соответственно, большое количество медленных соединений не вызовет никаких проблем. *WebFlux* продолжит принимать другие соединения, не приостанавливаясь в ожидании получения полного тела запроса. Абстракция реактивных потоков позволяет обрабатывать данные по мере их поступления. То есть сервер может контролировать обработку событий в зависимости от готовности сети.

Потоковые системы или системы реального времени

Еще одна область применения *WebFlux* – создание потоковых систем или систем реального времени. Чтобы понять, как *WebFlux* может помочь в этом, вспомним, какие системы называются потоковыми или реального времени.

Прежде всего такие системы характеризуются низкой задержкой и высокой пропускной способностью. В потоковых системах основной объем данных течет в направлении от сервера к клиенту, поэтому клиент играет роль потребителя. Обычно на стороне клиента случается меньше событий, чем на сервере. Однако в системах реального времени, таких как онлайн-игры, объем входящих данных примерно равен объему исходящих данных.

Низкой задержки и высокой пропускной способности можно добиться, используя неблокирующие взаимодействия. Как мы узнали в предыдущих разделах, неблокирующие и асинхронные взаимодействия обеспечивают эффективное исполь-

зование ресурсов. Самую высокую пропускную способность и низкую задержку демонстрируют системы на основе Netty или аналогичных фреймворков. Однако такие реактивные фреймворки имеют свои недостатки, предлагая сложные модели взаимодействий, основанные на использовании каналов и обратных вызовов.

Тем не менее реактивное программирование элегантно решает обе эти проблемы. Как мы узнали в главе 4 «*Project Reactor – основа реактивных приложений*», реактивное программирование, особенно с применением реактивных библиотек, таких как Reactor 3, помогает создавать асинхронные и неблокирующие конвейеры, лишь немного усложняя код и делая чуть более крутой кривую обучения. Оба решения включены в WebFlux. Используя Spring Framework, мы можем легко конструировать такие системы.

WebFlux в действии

Чтобы понять, как можно использовать WebFlux в реальных сценариях, напишем простенькое веб-приложение, подключающееся к удаленному Gitter Streams API через WebClient, передающее данные с помощью Project Reactor API и затем транслирующее сообщения во внешний мир через SSE. Общая структура системы изображена на рис. 6.26.

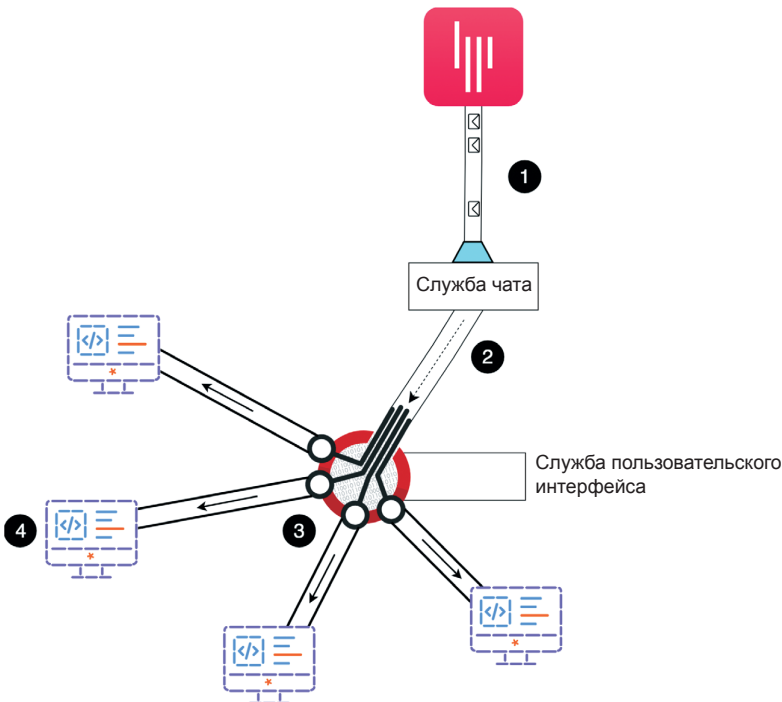


Рис. 6.26. Схематическая структура потокового приложения

Пояснения к схеме.

1. Точка интеграции с Gitter API. Как показано на рис. 6.26, между нашим сервером и службой Gitter осуществляются потоковые взаимодействия. Соответственно, в эту схему прекрасно вписывается реактивное программирование.
2. Точка в системе, где обрабатываются входящие сообщения и преобразуются в другое представление.
3. Точка, где принятые сообщения кешируются и рассылаются подключенным клиентам.
4. Подключенные браузеры.

Как видите, система имеет четыре центральных компонента. Чтобы создать эту систему, определим следующие классы и интерфейсы:

- ChatService: интерфейс, ответственный за связь с удаленным сервером, – он позволяет принимать сообщения от этого сервера;
- GitterService: реализация интерфейса ChatService, выполняющая подключение к потоковому Gitter API для приема новых сообщений;
- InfoResource: класс, реализующий обработку пользовательских запросов и отвечающий потоком сообщений.

Первый шаг к реализации системы – анализ интерфейса ChatService. Ниже показаны обязательные методы.

```
interface ChatService<T> {
    Flux<T> getMessagesStream();
    Mono<List<T>> getMessagesAfter(String messageId);
}
```

Этот интерфейс охватывает минимально необходимую функциональность, связанную с приемом сообщений. Метод `getMessagesStream` возвращает бесконечный поток сообщений в чате, а `getMessagesAfter` позволяет получить список сообщений с конкретным идентификатором.

В обоих случаях Gitter предлагает доступ к сообщениям через HTTP. То есть мы можем использовать простой `WebClient`. Следующий фрагмент демонстрирует, как можно реализовать `getMessagesAfter` и получить доступ к удаленному серверу.

```
Mono<List<MessageResponse>> getMessagesAfter(                //
    String messageId                                         //
) {                                                           //
    ...                                                       //
    return webClient                                         // (1)
        .get()                                               // (2)
        .uri(...)                                             // (3)
```

```

        .retrieve() // (4)
        .bodyToMono( // (5)
            new ParameterizedTypeReference
                <List<MessageResponse>>() {}
        ) //
        .timeout(Duration.ofSeconds(1)) // (6)
        .retryBackoff(Long.MAX_VALUE, Duration.ofMillis(500)); // (7)
    }

```

Этот пример показывает, как организовать простое взаимодействие «запрос-ответ» со службой Gitter. В (1) используем экземпляр WebClient, чтобы послать HTTP-запрос GET (2) удаленному серверу Gitter (3). Затем в строке (4) получаем ответ и преобразуем его с помощью WebClient DSL в поток Mono списков List из MessageResponse (5). Чтобы обеспечить устойчивую связь с внешней службой, определяем тайм-аут в строке (6) и в случае ошибки требуем повторения попытки (7).

Взаимодействия с потоковым Gitter API реализуются просто. Следующий фрагмент показывает, как подключиться к потоковой конечной точке JSON (*application/stream+json*) сервера Gitter.

```

public Flux<MessageResponse> getMessagesStream() { //
    return webClient //
        .get() // (1)
        .uri(...) //
        .retrieve() //
        .bodyToFlux(MessageResponse.class) // (2)
        .retryBackoff(Long.MAX_VALUE, Duration.ofMillis(500)); //
} //

```

Здесь мы используем тот же API (1). Единственное отличие – адрес URI, который мы скрыли, и тот факт, что теперь мы отображаем поток Flux, а не Mono, как можно видеть в строке (2). За кадром WebClient использует Decoder, доступный в контейнере. Это позволяет преобразовывать элементы из бесконечного потока на лету, не дожидаясь его конца.

Наконец, чтобы объединить оба потока в один и кешировать их, можно использовать следующий код, реализующий обработчик InfoResource.

```

@RestController // (1)
@RequestMapping("/api/v1/info") //
public class InfoResource { //

    final ReplayProcessor<MessageVM> messagesStream // (2)
        = ReplayProcessor.create(50); //

    public InfoResource( // (3)
        ChatService<MessageResponse> chatService //
    ) {
    }
}

```

```

    ) {
        Flux.mergeSequential(
            chatService.getMessageAfter(null)
            .flatMapIterable(Function.identity())
            chatService.getMessagesStream()
        )
        .map(...)
        .subscribe(messagesStream);
    }

    @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<MessageResponse> stream() {
        return messagesStream;
    }
}

```

Пояснения к коду.

1. Объявление класса с аннотацией `@RestController`.
2. Объявление поля `ReplayProcessor`. Как рассказывалось в главе 4 «*Project Reactor – основа реактивных приложений*», `ReplayProcessor` позволяет кешировать predetermined число элементов и пересылать последние элементы всем новым подписчикам.
3. Объявление конструктора класса `InfoResource`. Внутри конструктора мы определяем конвейер обработки, который объединяет поток последних сообщений от службы Gitter (строки 3.1 и 3.2). Если передан пустой идентификатор, Gitter вернет 30 последних сообщений. Также конвейер обработки принимает новые сообщения почти в режиме реального времени, как показано в строке (3.3). Затем все сообщения отображаются в модель представления (3.4) и на поток тут же подписывается `ReplayProcessor`. Это означает, что сразу после создания компонента `InfoResource` мы подключаемся к службе Gitter, кешируем последние сообщения и начинаем прием новых. Обратите внимание, что `mergeSequential` подписывается сразу на два потока данных, но отправлять сообщения из второго потока начинает только после исчерпания первого потока. Поскольку первый поток является конечным, мы получаем последние сообщения, а затем начинаем рассылать сообщения из `getMessagesStream`.
4. Объявление метода-обработчика, который вызывается для каждого нового соединения к заданной конечной точке. Здесь мы можем просто вернуть экземпляр `ReplayProcessor` (4.1), чтобы отослать сначала последние кешированные сообщения, потом начать отправку новых сообщений по мере их поступления.

Как видите, реализация такой сложной функциональности, как объединение двух потоков данных в требуемом порядке и кеширование 50 последних сообщений с последующей рассылкой их всем подписчикам, не требует больших усилий. Биб-

лиотека *Reactor* и модуль *WebFlux* берут на себя все самое сложное, и нам остается только написать бизнес-логику. Они обеспечивают эффективный неблокирующий ввод/вывод, а это значит, что с их помощью мы можем добиться высокой пропускной способности и низкой задержки.

Заключение

В этой главе мы узнали, что *WebFlux* можно рассматривать как эффективную замену старому доброму фреймворку *Web MVC*. Мы также узнали, что *WebFlux* использует те же приемы для объявления обработчиков запросов (с использованием хорошо известных аннотаций `@RestController` и `@Controller`). Кроме стандартного способа объявления обработчика *WebFlux* предлагает возможность функционального объявления легковесных конечных точек с использованием *RouterFunction*. Долгое время современные реактивные веб-серверы, такие как *Netty* и *Undertow*, были недоступны пользователям *Spring Framework*, но благодаря веб-фреймворку *WebFlux* мы получили возможность использовать их с применением уже знакомого API. Поскольку *WebFlux* основан на асинхронной и неблокирующей модели взаимодействий, этот фреймворк зависит от библиотеки *Reactor 3*, которая является базовым компонентом модуля.

Мы также исследовали изменения, обусловленные появлением нового модуля *WebFlux*. К ним относятся изменения во взаимодействиях между пользователями и сервером, которые теперь опираются на реактивные типы из *Reactor 3*, а также изменения во взаимодействиях между сервером и внешними службами, в частности с использованием новой технологии *WebClient* и новый *WebSocketClient*, обеспечивающий клиент-серверные взаимодействия через *WebSocket*. Кроме того, *WebFlux* является кросс-библиотечным фреймворком, то есть он поддерживает любые библиотеки, совместимые со стандартом *Reactive Streams*, и может использовать их вместо библиотеки по умолчанию *Reactor 3*.

Затем в этой главе было представлено детальное сравнение *WebFlux* и *Web MVC* с разных точек зрения. Подводя итог, можно сказать, что в большинстве случаев *WebFlux* является верным выбором для реализации высоконагруженных веб-серверов и по всем показателям имеет двукратное преимущество перед *Web MVC*. Мы рассмотрели, какие преимущества дает бизнесу использование модуля *WebFlux*, и выяснили, как он упрощает работу. Познакомились также с недостатками этой технологии.

Наконец, мы освоили несколько сценариев, в которых *WebFlux* является наиболее подходящим решением. К ним относятся системы на основе микросервисов, потоковые системы реального времени, онлайн-игры и другие похожие области применения, где наиболее важными характеристиками являются низкая задержка, высокая пропускная способность, низкое потребление памяти и эффективное использование процессора.

Мы познакомились со многими ключевыми аспектами веб-приложений, но не затронули еще одну важную сторону – взаимодействие с базами данных. В следующей главе рассмотрим основные особенности реактивной связи с базами данных, выясним, какие базы данных поддерживают реактивные взаимодействия, и определим, что делать, когда поддержка реактивности отсутствует.

Глава 7

Реактивный доступ к базам данных

В предыдущей главе мы познакомились с новым пополнением в семействе Spring Framework – модулем Spring WebFlux. Он обеспечивает возможность реализации реактивных программных интерфейсов приложений и неблокирующей обработки HTTP-запросов всех видов.

В этой главе узнаем, как получить реактивный доступ к данным с использованием модулей *Spring Data*. Это жизненно важно для создания полностью реактивного и отзывчивого приложения, которое максимально эффективно использует все доступные вычислительные ресурсы, обеспечивая высокую ценность для бизнеса и одновременно требуя незначительных эксплуатационных расходов.

Даже если выбранная база данных не имеет реактивного или асинхронного драйвера, все равно сохраняется возможность создать реактивное приложение, используя выделенный пул потоков выполнения, – глава расскажет, как сделать это. Тем не менее ни в каких реактивных приложениях нельзя рекомендовать использование блокирующего ввода/вывода.

Будут рассмотрены следующие темы:

- модели хранения и обработки данных в современном мире;
- достоинства и недостатки асинхронного доступа к данным;
- как Spring Data обеспечивает реактивный доступ к данным в реактивных приложениях;
- какие реактивные коннекторы доступны в настоящее время;
- как адаптировать блокирующий ввод/вывод к модели реактивного программирования.

Модели обработки данных в современном мире

Даже притом что монолитные программные системы продолжают существовать, работать и поддерживать решение многих повседневных задач, большинство новых систем проектируется или, по крайней мере, планируется с прицелом на архитектуру микросервисов. В настоящее время микросервисы являются, пожалуй, доминирующим архитектурным стилем для современных приложений, особенно действующих в *облачном окружении*. В большинстве случаев такой архитектурный стиль обеспечивает высокую скорость разработки программного обеспечения. В то же время он предлагает возможность создания более эффективной с точки зрения затрат базовой инфраструктуры (серверы, сети, резервное копирование и т. д.), особенно когда он опирается на услуги облачных провайдеров, таких как AWS, Google Cloud Platform или Pivotal Cloud Foundry.



За дополнительной информацией об облачных приложениях обращайтесь к хартии фонда **Cloud Native Computing Foundation (CNCF)**: <https://cncf.io/about/charter>. Подробнее о достоинствах и недостатках облачных приложений в контексте реактивного программирования рассказывается в главе 10 «И наконец, выпуск!».

А теперь рассмотрим основы хранения данных с точки зрения микросервисов, возможные стратегии, подходы к реализации и некоторые рекомендации.

Предметно-ориентированное проектирование

Книга Эрика Эванса (Eric Evans) «*Domain-Driven Design. Tackling Complexity in the Heart of Software*»³ (Addison-Wesley, 2004) должна быть на полке у каждого разработчика программного обеспечения, потому что в ней определяется и формализуется важная теоретическая основа для успешной архитектуры микросервисов. DDD устанавливает общий словарь (а именно контекст, предметную область, модель и единый язык) и формулирует набор принципов для поддержания целостности модели. Один из важнейших выводов DDD заключается в том, что отдельные **ограниченные контексты**, определяемые в терминах DDD, обычно отображаются в отдельные микросервисы, как показано на рис. 7.1.

³ Эванс Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. М.: Вильямс, 2010. – Прим. перев.



Рис. 7.1. Ограниченные контексты (кандидаты на реализацию в виде микросервисов), как их обозначает Вон Вернон (Vaughn Vernon), автор книги «*Implementing Domain-Driven Design and Domain-Driven Design Distilled*»⁴

Поскольку предметно-ориентированное проектирование фокусируется на предметной области бизнеса, особенно на артефактах, помогающих выражать, создавать и извлекать предметные модели, в этой главе мы часто будем использовать следующие понятия: *сущность*, *объект значения*, *агрегат*, *репозиторий*.



Узнать больше о понятиях DDD можно в статье http://dddcommunity.org/resources/ddd_terms.

В процессе реализации приложения, спроектированного с учетом предметно-ориентированных особенностей, предыдущие понятия должны отображаться в слой хранения внутри приложения, если он присутствует. Такая предметная модель служит основой для логических и физических моделей данных.

Хранение данных в эпоху микросервисов

Одним из важнейших изменений в хранении данных, которые принесла с собой архитектура микросервисов, является, пожалуй, отказ от использования разными службами общего хранилища. То есть каждая логически отдельная служба должна иметь свою базу данных (если она вообще необходима) и в идеале никакие другие службы не должны иметь доступа к данным, иначе как через API службы-владельца.

Объяснение причин такого разделения выходит далеко за рамки этой книги, но мы отметим наиболее важные из них:

⁴ Вернон В. Реализация методов предметно-ориентированного проектирования. М.: Вильямс, 2017. – Прим. перев.

- возможность развивать разные службы независимо друг от друга, без тесной связи со схемами данных;
- теоретически более точное управление ресурсами;
- возможность горизонтального масштабирования;
- возможность в каждом случае использовать наиболее подходящее решение хранения данных.

Рассмотрим диаграмму на рис. 7.2.

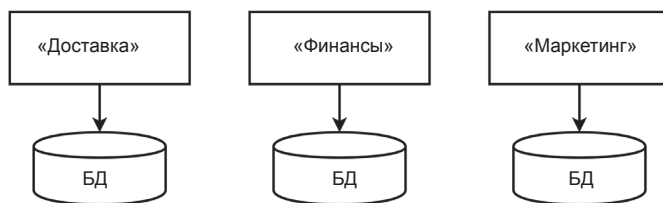


Рис. 7.2. Каждая служба имеет отдельную базу данных

На физическом уровне разделить хранилища можно разными способами. Самый простой – запустить один сервер баз данных и одну базу данных для всех служб, но для каждой определить свою отдельную схему данных (*своя схема для каждого микросервиса*). Такая конфигурация легко реализуется, потребляет минимальный объем ресурсов сервера и не требует больших усилий для администрирования во время эксплуатации, поэтому выглядит очень привлекательно для начального этапа разработки приложения. Разделение данных можно также организовать на уровне правил, управляющих доступом к базе данных. Такой подход легко реализуется, но в то же время легко нарушается. Так как данные хранятся в одной базе данных, у разработчиков может возникнуть соблазн написать единый запрос, способный извлекать или изменять данные, принадлежащие нескольким службам. Кроме того, чтобы взломать всю систему целиком, достаточно взломать только одну службу. На рис. 7.3 показано, как выглядит архитектура системы, построенной по принципу *своя схема для каждого микросервиса*.

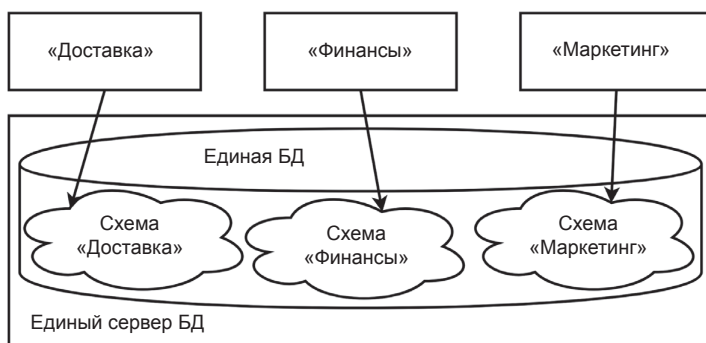


Рис. 7.3. *Своя схема для каждой службы*

Службы могут совместно пользоваться одним сервером баз данных, но должны иметь разные базы данных с разными учетными записями. Такой подход улучшает отделение данных, потому что намного сложнее написать единый запрос, способный получить доступ к внешним данным. Правда, такое решение немного усложняет процедуру резервного копирования. Этот подход изображен на рис. 7.4.

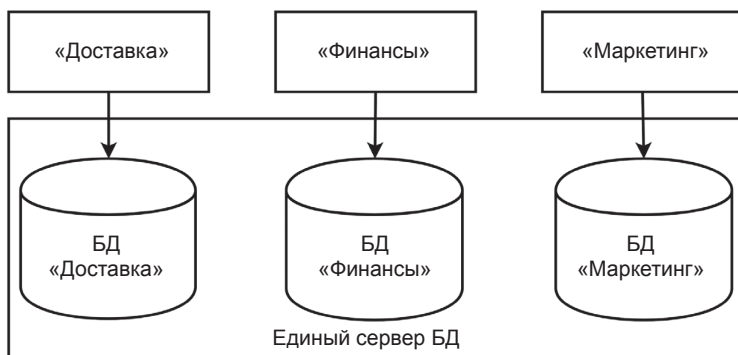


Рис. 7.4. Своя база данных для каждой службы

Для каждой службы можно запустить свой сервер баз данных. Такой подход требует больше затрат на администрирование, но обеспечивает хорошую начальную точку для тонкой настройки сервера баз данных, лучше отвечающего потребностям конкретной службы. Также в этом случае возможно вертикальное и горизонтальное масштабирование. Данный подход изображен на рис. 7.5.

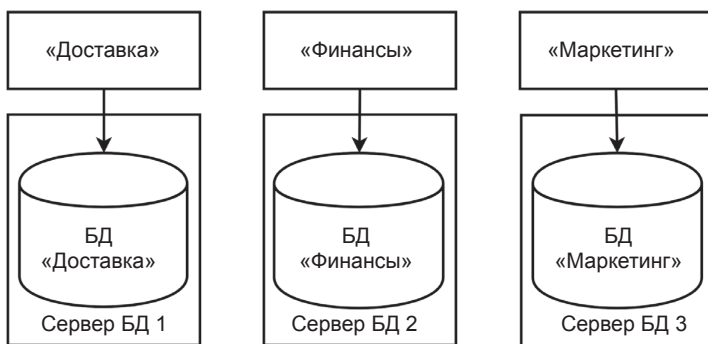


Рис. 7.5. Свой сервер баз данных для каждой службы

При реализации программной системы с успехом можно использовать сразу все три перечисленных подхода в разных пропорциях в зависимости от конкретных потребностей системы. Схема на рис. 7.6 иллюстрирует такую систему.

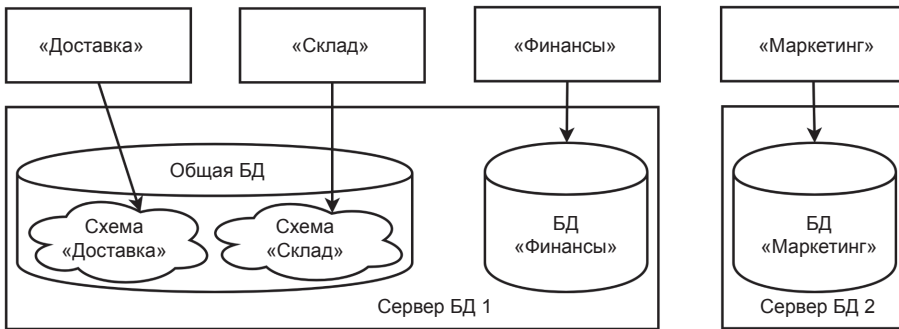


Рис. 7.6. Комбинированная стратегия хранения

Более того, для достижения лучших результатов можно использовать разные реализации серверов баз данных и разные системы управления базами данных (например, SQL и NoSQL). Этот подход называют **polyglot persistence** (использование хранилищ разного типа).

Использование хранилищ разного типа

В 2006 году Нил Форд (Neal Ford) предложил термин **polyglot programming** (многоязычное программирование). Он выражает идею создания программных систем сразу на нескольких языках программирования, чтобы получить дополнительные выгоды от использования наиболее подходящего языка в том или ином техническом или бизнес-контексте. После этого появилось много новых языков программирования, созданных с целью лучше удовлетворить потребности в разных областях.

Одновременно произошла похожая смена мировоззрения в сфере хранения данных. Это многих заставило задуматься над вопросом: что получится, если разные части приложения будут использовать разные технологии хранения данных в зависимости от технических и бизнес-требований? Например, технологии хранения HTTP-сеансов в распределенных веб-приложениях и графа друзей в социальной сети должны обладать разными рабочими характеристиками и, следовательно, должны быть основаны на использовании разных баз данных. В настоящее время для большинства систем достаточно использовать одновременно две разные базы данных или более.

Исторически сложилось так, что многие **системы управления реляционными базами данных** (СУРБД) построены на одних и тех же принципах ACID (Atomicity, Consistency, Isolation, Durability – атомарность, согласованность, изолированность, долговечность) и предлагают довольно похожие диалекты языка SQL для взаимодействия с хранилищем. В общем случае СУРБД хорошо подходят для широкого круга приложений, но они редко показывают лучшую производительность и рабочие характеристики в некоторых распространенных сценариях

(например, для хранения графов, для организации хранилищ в оперативной памяти и для распределенных хранилищ). Базы данных **NoSQL**, появившиеся в последние годы, напротив, основаны на более широком спектре основополагающих принципов, что повышает их привлекательность в некоторых особых случаях, но базы данных NoSQL редко оказываются эффективным решением в роли универсальных хранилищ. Это обстоятельство подчеркивает диаграмма на рис. 7.7.

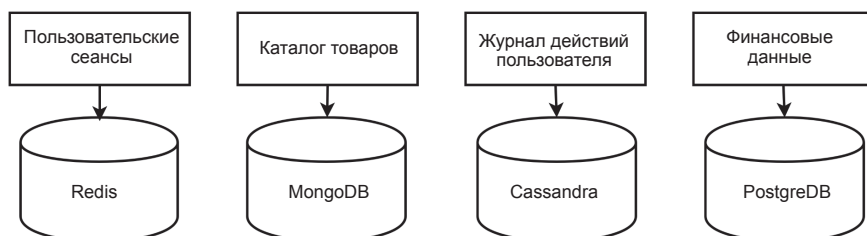


Рис. 7.7. Каждая служба использует технологию хранения, лучше отвечающую ее потребностям

С другой стороны, использование хранилищ разного типа влечет дополнительные затраты и сложности. Каждый новый механизм хранения требует освоения новых API и парадигм, разработки или адаптации новых клиентских библиотек, а также решения нового набора стандартных проблем в разработке и эксплуатации. Кроме того, неправильное использование базы данных NoSQL может означать полное перепроектирование службы. Использование правильной технологии хранения (SQL или NoSQL) несет определенные удобства, но не избавляет от проблем.

В Spring Framework имеется отдельный проект, имеющий отношение к хранению данных, который называется Spring Data (<http://projects.spring.io/spring-data>). Оставшаяся часть главы будет посвящена разным архитектурным подходам и способам подключения к базам данных, доступным в Spring Data, особенно ориентированным на модели реактивного программирования, и тому, как они меняют способы доступа приложений к данным, хранимым в слое хранилищ разного типа.

База данных как услуга

В правильно спроектированной архитектуре микросервисов ни одна из служб не имеет состояния, и все состояния хранятся в особых службах, которые знают, как управлять хранением данных. В облачном окружении службы без состояния эффективнее обеспечивают масштабируемость и высокую доступность. Однако масштабировать серверы баз данных намного сложнее, особенно если они не предназначены для использования в облаке. Большинство поставщиков облачных услуг предлагает решение этой проблемы в виде своих решений **DBaaS** (Database as a Service – база данных как услуга). Такие решения могут быть специально настроенными версиями обычных баз данных (MySQL, PostgreSQL и Redis) или изначально спроектированными для работы только в облаке (AWS Redshift, Google BigTable и Microsoft CosmosDB).

Обычно алгоритм использования облачного хранилища или базы данных достаточно прост.

1. Клиент посылает запрос на доступ к базе данных или хранилищу (через страницу администрирования или посредством API).
2. Поставщик облачных услуг открывает доступ к API или серверному ресурсу, который можно использовать для сохранения данных. При этом клиент не знает и не должен знать, как реализован этот API.
3. Клиент использует API хранилища или драйвер базы данных, указывая свои учетные данные.
4. Поставщик облачных услуг взимает плату с клиента в зависимости от тарифа подписки, объема хранимых данных, частоты запросов, наличия одно-временных соединений и других характеристик.

Обычно такой подход позволяет клиенту (в нашем случае разработчику) и поставщику облачных услуг сосредоточиться на решении основных задач. Поставщик облачных услуг реализует наиболее эффективный способ хранения и обработки данных клиента, минимизирующий затраты на базовую инфраструктуру, а клиент фокусирует свое внимание на бизнес-целях приложения и не тратит времени на настройку серверов баз данных, репликацию или создание резервных копий. Такое разделение интересов не всегда выгодно для клиента, а иногда даже просто невозможно. Однако, когда это уместно, подобный подход позволяет создавать успешные и популярные приложения усилиями небольшой группы инженеров.



Можно привести в пример компанию Foursquare, чьи приложения используют более 50 млн человек в месяц и построены с применением стека технологий AWS, а именно Amazon EC2 для облачного хостинга, Amazon S3 для хранения изображений и других данных и Amazon Redshift в роли базы данных.

Вот несколько хорошо известных облачных хранилищ данных и услуг баз данных:

- **AWS S3** – хранилище пар ключ/значение, доступное через веб-интерфейс (REST API или AWS SDK). Предназначено для хранения файлов, изображений, резервных копий и любой другой информации, которую можно представить как массив байтов;
- **AWS DynamoDB** – полностью управляемая коммерческая база данных NoSQL, обеспечивающая возможность синхронной репликации с несколькими вычислительными центрами;
- **AWS Redshift** – хранилище данных, базирующееся на технологии параллельной обработки (MPP). Поддерживает возможность анализа больших данных;
- **Heroku PostgreSQL as a Service** – база данных PostgreSQL, поддерживаемая поставщиком облачных услуг Heroku, который предлагает общие и эксклюзивные серверы баз данных для приложений, развернутых в кластере Heroku;

- **Google Cloud SQL** – полностью управляемые базы данных PostgreSQL и MySQL, предлагаемые компанией Google;
- **Google BigTable** – высокопроизводительное и коммерческое хранилище данных, предназначенное для обработки больших объемов данных в приложениях с низкой задержкой и высокой пропускной способностью;
- **Azure Cosmos DB** – коммерческая глобально-распределенная база данных от Microsoft, поддерживающая смешанную систему моделей. Имеет несколько разных API, включая поддержку протокола на уровне драйвера MongoDB.

Разделение данных между микросервисами

В реальных бизнес-системах для обработки клиентского запроса часто приходится запрашивать данные, принадлежащие двум или более службам. Например, клиенту может понадобиться увидеть все свои заказы и признак их оплаты. До появления архитектуры микросервисов это можно было реализовать одним запросом, но теперь это противоречит правилам. Для обработки запроса, вовлекающего обращение к нескольким службам, необходимо реализовать службу-адаптер, которая запросит две другие службы, отвечающие за заказы и платежи, применит все необходимые преобразования и вернет клиенту объединенный результат. Кроме того, совершенно очевидно, что если две службы тесно связаны друг с другом и в значительной степени взаимозависимы, это может служить сигналом к объединению служб (если такое объединение не навредит предметно-ориентированному дизайну). Данный подход отражает диаграмма на рис. 7.8.

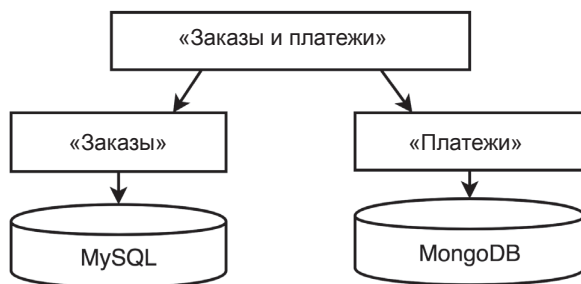


Рис. 7.8. Служба-адаптер, объединяющая данные из служб «Заказы» и «Платежи»

Стратегия чтения реализуется просто, намного сложнее реализовать стратегию изменения данных, вовлекающую сразу несколько служб. Допустим, клиент хочет сделать заказ, но это возможно только после проверки запасов на складе и получения подтверждения оплаты. Каждая служба имеет свою базу данных, поэтому в бизнес-транзакции участвуют два микросервиса и базы данных или более. Есть несколько подходов к решению этой задачи, но наибольшей популярностью пользуются распределенные транзакции и архитектуры, управляемые событиями.

Распределенные транзакции

Распределенная транзакция – это транзакция, которая изменяет данные в двух или более компьютерных системах, подключенных к сети. Иначе говоря, несколько служб, действующих в комплексе, должны прийти к согласию о том, что некоторое действие состоялось или не состоялось. На практике большинство баз данных использует строгую и двухфазную блокировку, чтобы гарантировать глобальную согласованность.

Распределенные транзакции часто применяются службами для атомарного изменения хранимых данных. Они нередко использовались в монолитных приложениях, чтобы гарантировать надежное выполнение некоторых действий в разных хранилищах данных. Это также способствовало правильному восстановлению после сбоев. Однако в настоящее время не рекомендуется прибегать к распределенным транзакциям между несколькими микросервисами. Это обусловлено несколькими причинами. Наиболее важные перечислены ниже:

- служба, поддерживающая распределенные транзакции, должна иметь API двухфазного подтверждения, реализовать который очень непросто;
- микросервисы, вовлекаемые в распределенную транзакцию, оказываются чересчур тесно связанными, что явно противоречит идее архитектуры на основе микросервисов;
- распределенные транзакции не масштабируются, ограничивают пропускную способность системы и соответственно ухудшают масштабируемость системы.

Событийно-ориентированные архитектуры

Лучший способ реализовать поддержку распределенных бизнес-транзакций в окружении микросервисов – создать событийно-ориентированную архитектуру, которую мы уже несколько раз рассматривали в этой книге.

Если требуется изменить состояние системы, первая служба изменяет свои данные в своей базе данных и в рамках той же самой внутренней транзакции посылает событие брокеру сообщений. То есть транзакции не пересекают границ служб. Вторая служба, подписавшаяся на требуемый тип событий, получает события и изменяет соответственно данные в своем хранилище и, возможно, посылает свое событие. Службы остаются независимыми друг от друга. Единственное, что их связывает, – это сообщения, которыми они обмениваются. В отличие от распределенных транзакций, событийно-ориентированная архитектура позволяет системе нормально действовать, даже если вторая служба не будет работать, когда первая выполнит действие. Это очень важная характеристика, потому что она напрямую влияет на устойчивость системы. Распределенная транзакция требует, чтобы все вовлеченные компоненты (микросервисы) были доступны и исправно работали, пока она протекает. Чем больше микросервисов в системе или чем

шире вовлеченность их в распределенные транзакции, тем сложнее такой системе действовать.

Так же как и раньше, когда две службы много общаются друг с другом, они могут рассматриваться как кандидаты на слияние. Кроме того, службы-адаптеры, выполняющие изменения в нескольких службах одновременно, тоже можно реализовать с применением событий.

Согласованность в конечном счете

Давайте оглянемся назад и посмотрим, какую роль играют распределенные транзакции в программной системе. Очевидно, что главная их цель – обеспечить пребывание системы в некотором определенном состоянии. Иначе говоря, они устраняют неопределенность, которая может быть вызвана несовместимостью состояний разных компонентов системы. Однако такое устранение неопределенности является очень ограничительным требованием. Вон Вернон (Vaughn Vernon), автор книги *«Implementing Domain-Driven Design and Domain-Driven Design Distilled»*⁵, предложил *внедрить неопределенность в предметную модель*. По его словам, если систему сложно защитить от противоречивого состояния, которое все равно возникает, сколько бы мы с ним ни боролись, выгоднее будет принять неопределенность и сделать ее частью обычного бизнес-процесса.

Например, наша система могла бы создавать заказ без подтверждения платежа, вводя новое состояние с названием **проверка платежной информации**. Это новое событие превращает неопределенное состояние (информация о платеже может подтвердиться или не подтвердиться) в отдельный бизнес-этап, требующий некоторого времени для завершения (пока информация о платеже не подтвердится). С таким подходом мы не требуем, чтобы система постоянно находилась в непротиворечивом состоянии. Мы лишь хотим, чтобы система имела непротиворечивое представление о каждой бизнес-транзакции. Такая непротиворечивость в будущем называется **согласованностью в конечном счете**. Суть ее иллюстрирует диаграмма на рис. 7.9.

⁵ Вернон В. Реализация методов предметно-ориентированного проектирования. М.: Вильямс, 2017. – Прим. перев.

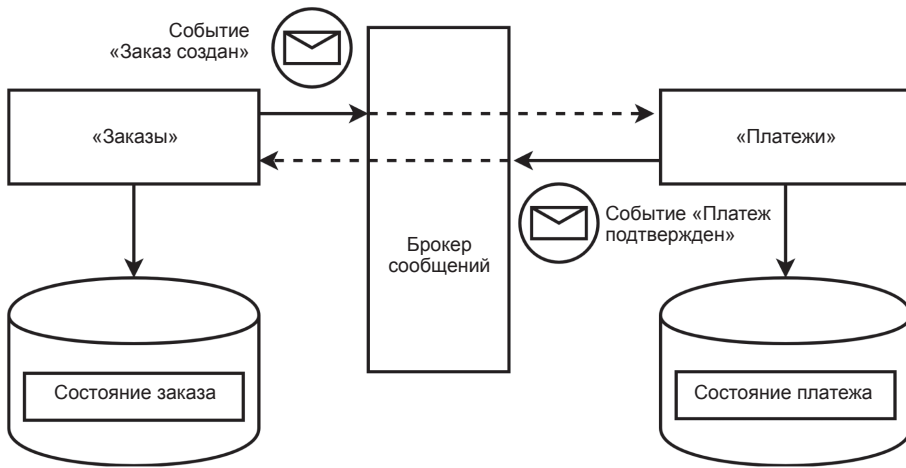


Рис. 7.9. Обе службы, «Заказы» и «Платежи», изменяют свои базы данных и сообщают о протекании рабочего процесса посредством сообщений

Обычно гарантий согласованности в конечном счете достаточно для создания надежной системы, успешно справляющейся со своей работой. Более того, любая распределенная система должна поддерживать согласованность в конечном счете, чтобы оставаться доступной (обрабатывать запросы пользователей) и устойчивой к разделению (поддерживать восстановление нормальной работоспособности после сбоев в работе сети, связывающей компоненты).

Шаблон SAGA

Один из самых популярных шаблонов для реализации распределенных транзакций, особенно в мире микросервисов, называется **SAGA** (сага). Он был придуман в 1987 году для управления долгоживущими транзакциями в базах данных.

Сага состоит из нескольких мелких транзакций, каждая из которых является локальной для своего микросервиса. В этом случае внешний запрос инициирует сагу и запускает первую небольшую транзакцию, которая при успешном завершении запускает вторую транзакцию и т.д. Если какая-то транзакция в середине терпит неудачу, запускается действие, компенсирующее предыдущие транзакции. Существует два основных способа реализации шаблона – *посредством регистрации событий и обращением к службе-координатору*.

Регистрация событий

Обрабатывая события, протекающие через приложение, состоящее из набора микросервисов, микросервисы могут использовать шаблон **регистрации событий** (event sourcing). В этом случае состояние бизнес-сущности хранится как последовательность событий, изменяющих его. Например, банковский счет может быть

представлен начальной суммой и последовательностью операций списания/пополнения. Наличие этой информации не только позволяет вычислить текущее состояние счета, воспроизведя все события, изменяющие его, но и дает надежный журнал, отражающий порядок изменения сущности и позволяющий определить это состояние в любой момент в прошлом. Обычно службы, реализующие шаблон регистрации событий, предлагают API, с помощью которого другие службы могут подписаться на получение событий, изменяющих сущность.

Чтобы оптимизировать время, необходимое для вычисления текущего состояния, приложение может периодически создавать и сохранять мгновенные снимки (snapshots). А чтобы уменьшить объем хранимых данных, все события, предшествующие снимку, могут удаляться. Очевидно, что в этом случае часть истории развития сущности теряется.

Вот пример журнала событий для банковского счета 111-11.

Дата	Операция	Сумма
2018-06-04 22:00:01	Создание	0
2018-06-05 00:05:00	Пополнение	50
2018-06-05 09:30:00	Списание	10
2018-06-05 14:00:30	Пополнение	20
2018-06-06 15:00:30	Пополнение	115
2018-06-07 10:10:00	Списание	40

Текущий баланс: 135

Несмотря на простоту, регистрация событий редко используется на практике из-за малой известности и немного чуждого подхода к программированию, а также из-за слишком крутой кривой обучения. Кроме того, из-за необходимости постоянно пересчитывать состояние метод регистрации событий не обеспечивает высокой эффективности обработки запросов, особенно если запросы очень сложные. В этом случае поможет шаблон разделения ответственности на команды и запросы (Command Query Responsibility Segregation).

Разделение ответственности на команды и запросы

Разделение ответственности на команды и запросы (Command Query Responsibility Segregation, CQRS) часто используется в паре с регистрацией событий. Шаблон CQRS состоит из двух частей:

- **пишущая** часть получает команды изменения состояния и сохраняет их в базовом хранилище событий, но не возвращает состояния сущности;
- **читающая** часть не изменяет состояния сущности и возвращает его в ответ на запросы. Представления состояний для отдельных запросов хранят-

ся отдельно и пересчитываются асинхронно после получения изменяющих событий.

Принцип действия шаблона CQRS изображен на рис. 7.10.

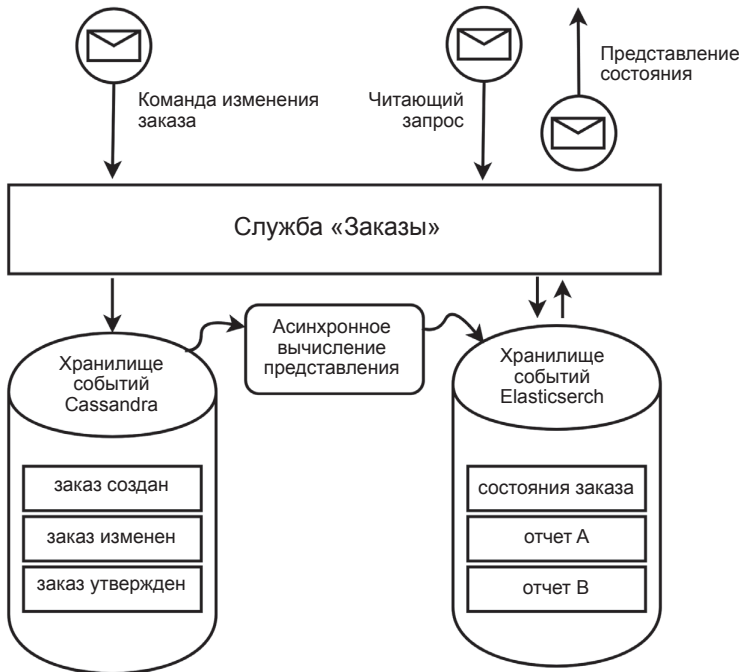


Рис. 7.10. Реализация шаблона CQRS для службы «Заказы». Пишущая часть сохраняет изменяющие команды, а читающая часть асинхронно вычисляет представления для ожидаемых запросов

Шаблон CQRS позволяет программным системам обрабатывать огромные объемы данных потоковым способом и в то же время быстро отвечать на разные запросы, касающиеся текущего состояния системы.

Бесконфликтно реплицируемые типы данных

Чем больше становится наше приложение, тем больше данных оно должно обрабатывать, даже если имеет единственный микросервис с четко ограниченной областью. Как упоминалось выше, транзакции не поддаются масштабированию, и с ростом приложения становится все труднее поддерживать глобальную целостность состояния даже в рамках одного микросервиса. Поэтому для лучшей производительности и масштабируемости системы мы можем разрешить конкурентное изменение данных разными экземплярами службы, отказавшись от глобальной блокировки или согласования транзакций. Такой подход называется **оптимистической репликацией** и позволяет репликам данных развиваться **параллельно** с возможными несоответствиями, которые должны быть улаже-

ны позже. В таких сценариях несогласованность между репликами устраняется на этапе их слияния. В этот момент все конфликты между ними должны быть ликвидированы, но иногда для этого требуется отменить некоторые изменения, что может быть неприемлемо с точки зрения пользователя. Однако существуют такие структуры данных, обладающие особыми математическими свойствами, которые гарантируют успех слияния реплик. Подобные структуры называются **бесконфликтно реплицируемыми типами данных** (Conflict-Free Replicated Data Types, CRDT).

CRDT – это типы данных, которые могут реплицироваться по нескольким вычислительным единицам, изменяться конкурентно без всякой координации и затем объединяться для приведения в целостное, непротиворечивое состояние. Эта идея была описана в 2011 году Марком Шапиро (Marc Shapiro), Нуно Прегуиком (Nuno Preguica), Марекком Завирски (Marek Zawirski) и Карлосом Бакеро (Carlos Baquero). На момент написания этих строк к CRDT относилось несколько типов данных, таких как «только растущий счетчик», «только расширяющееся множество», «двухфазное множество» Two-Phase Set, «множество – последний записанный побеждает» и некоторые другие множества, которые могут удовлетворить лишь час бизнес-требований. Однако типы CRDT полезны для сценариев совместного редактирования текста, онлайн-чатов и азартных онлайн-игр. SoundCloud, платформа распространения звукозаписей, использует CRDT и веб-фреймворк Phoenix для поддержки обмена информацией между множеством узлов в режиме реального времени, а Microsoft Cosmos DB использует CRDT для записи данных. База данных Redis также имеет встроенную поддержку CRDT в форме **бесконфликтно реплицируемой базы данных** (Conflict-Free Replicated Database, CRDB).

Система обмена сообщениями как хранилище данных

Опираясь на идею регистрации событий, можно сделать вывод, что *брокер сообщений с хранилищем для них может уменьшить потребность в выделенной базе данных* для отдельных микросервисов. И действительно, если все события изменения сущности (включая мгновенные снимки) будут храниться брокером достаточно продолжительное время и будут доступны для чтения в любой момент, мы сможем определить состояние всей системы исключительно по событиям. В процессе запуска каждая служба сможет прочесть историю последних событий (с момента создания последнего мгновенного снимка) и вычислить состояние сущности в памяти. То есть служба сможет работать, просто обрабатывая новые команды на изменение и запросы на чтение, а также время от времени генерируя мгновенные снимки сущности и посылая их брокеру.

Apache Kafka – популярный распределенный брокер сообщений с надежным слоем хранения, его можно использовать как основное и, возможно, единственное хранилище данных в системе.

Как можно заметить, в настоящее время хранилища разного типа и событийно-ориентированные архитектуры, основанные на брокерах сообщений, часто используются в тандеме для реализации надежных и сложных процессов в масштабируемых и постоянно меняющихся программных системах. Оставшаяся часть главы будет посвящена механизмам хранения, предлагаемым Spring Framework, а в главе 8 «*Масштабирование с Cloud Streams*» вы узнаете, какие приемы доступны в экосистеме Spring для реализации эффективных приложений на основе событийно-ориентированной архитектуры.

Синхронная модель извлечения данных

Чтобы узнать все достоинства и недостатки реактивных подходов к хранению данных, вспомним, как был организован доступ к данным в дореактивную эпоху. Также посмотрим, как взаимодействуют клиент и база данных, обмениваясь запросами и ответами, какие части этих взаимодействий могут обрабатываться асинхронно и какие могли бы получить дополнительные выгоды от применения шаблонов реактивного программирования. Слой хранения данных состоит из нескольких уровней абстракции, и мы рассмотрим все эти уровни, примеряя к ним реактивный наряд.

Протокол связи для доступа к базе данных

Существует тип баз данных, которые называются **встроенными базами данных**. Такие базы данных выполняются внутри самого прикладного процесса и не требуют взаимодействовать с ними посредством сети. Для встроенных баз данных не действует жесткое требование иметь протокол связи, хотя некоторые имеют его или могут работать в двух режимах: встроенном и в виде отдельной службы. Далее в этой главе мы используем в нескольких примерах встроенную базу данных H2.

Однако в большинстве программ используются базы данных, выполняющиеся в отдельных процессах на отдельных серверах (или в отдельных контейнерах). Для взаимодействия с такой внешней базой данных приложение использует специальную клиентскую библиотеку, называемую **драйвером базы данных**. Кроме того, протокол связи определяет, как должны взаимодействовать между собой база данных и ее драйвер, и задает формат сообщений, пересылаемых между клиентом и базой данных. В большинстве случаев протокол связи не зависит от языка программирования, благодаря чему приложение на Java, например, может посылать запросы базе данных, написанной на C++.

Обычно протоколы связи предназначены для работы поверх TCP/IP, поэтому в них отсутствует необходимость использования блокировок. Так же как в случае синхронных взаимодействий по HTTP, сам протокол ничего не блокирует – клиент сам решает, когда ему заблокироваться в ожидании результатов. Более того,

TCP – асинхронный протокол, поддерживающий обратное давление через поток управления, реализуемый скользящим окном. Однако подход со скользящим окном предназначен для отправки блоков байтов по Сети и не всегда соответствует потребностям приложений управлять обратным давлением. Например, принимая записи из базы данных, намного естественнее запрашивать следующую порцию данных в количестве записей, а не полагаться на системные настройки, определяющие размер сетевого буфера. Конечно, для реализации обратного давления протокол связи может намеренно использовать другой механизм или даже комбинацию механизмов, но важно помнить, что механизмы TCP никуда не деваются и продолжают работать за кадром.

Также в качестве основы для протокола связи может использоваться протокол более высокого уровня, например HTTP2, WebSocket, gRPC или RSocket. В главе 8 «*Масштабирование с Cloud Streams*» вы найдете краткое сравнение протоколов RSocket и gRPC.

Помимо обратного давления для передачи больших наборов данных между клиентом и базой данных используются некоторые другие подходы. Например, клиент добавляет десятки тысяч записей или результат аналитического запроса включает миллионы записей. Для простоты рассмотрим только последний случай. В целом есть несколько подходов к реализации передачи такого большого набора данных:

- подготовить **весь набор результатов** на стороне базы данных, поместить данные в контейнер и послать контейнер целиком сразу по завершении обработки запроса. Этот подход не предполагает никакого обратного давления и требует выделять огромные буферы на стороне базы данных (а также, возможно, на стороне клиента). Кроме того, клиент получит первые результаты только после окончания обработки запроса. Такой подход легко реализуется, к тому же процесс выполнения запроса длится не особенно долго, и это может способствовать уменьшению конфликтов с запросами на изменение, которые могут поступить в то же время;
- посылать **результаты блоками** по запросу со стороны клиента. Запрос может быть выполнен полностью, и результаты сохранены в буфер. Как вариант база данных может выполнять запрос до того момента, когда окажется заполненным запрошенное число блоков, и возобновлять его выполнение только по требованию клиента. При такой организации работы может потребоваться меньше буферов в памяти, первые записи будут возвращаться еще до окончания выполнения запроса, и есть возможность управлять обратным давлением или прерывать выполнение запроса;
- посылать **результаты как поток данных** по мере их получения в ходе выполнения запроса. В таком случае клиент получает возможность сообщить базе данных, какой объем он готов принять и управлять обратным давлением, фактически влияя на процесс выполнения запроса. Такой подход почти не требует создания дополнительных буферов, и клиент получает первую

запись из результатов так быстро, насколько возможно. Однако этот подход может приводить к снижению эффективности использования сети и процессора из-за необходимости обмениваться существенным объемом управляющей информации и частых обращений к системным вызовам.

На рис. 7.11 показан процесс взаимодействий, когда результаты возвращаются блоками.

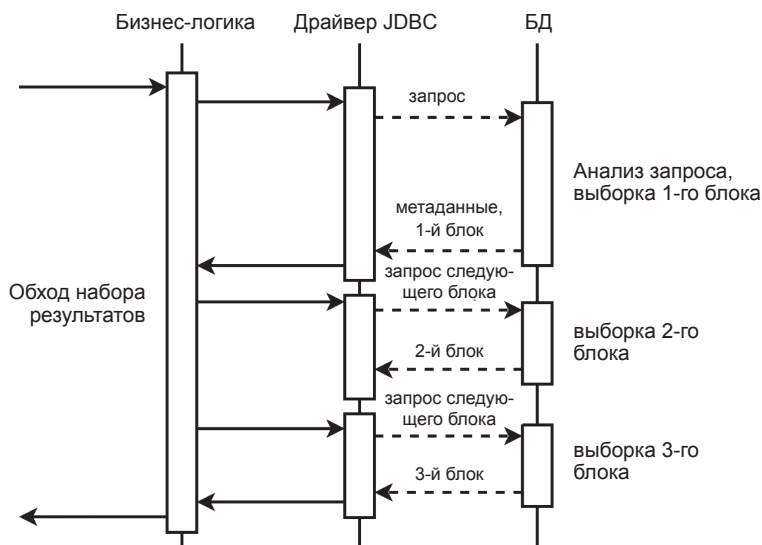


Рис. 7.11. Отправка результатов выполнения запроса блоками

Обычно базы данных реализуют один или несколько подходов в своих протоколах связи. Например, MySQL может вернуть результаты целиком или в виде потока записей. В то же время PostgreSQL реализует такое понятие, как **портал**, когда клиент может запросить передать ему столько записей, сколько он готов принять. На рис. 7.11 показано, как Java-приложение использует такой подход.

То есть хорошо спроектированный протокол связи, реализуемый базой данных, уже может включать все, что требуется для реактивности. В то же время даже самый простой протокол можно обернуть реактивным драйвером, управляющим обратным давлением через поток управления TCP.

Драйвер базы данных

Драйвер базы данных – это библиотека, приспособливающая протокол связи с базой данных к языковым конструкциям, таким как вызовы методов, обратные вызовы или механизмы поддержки реактивных потоков. Драйверы реляционных баз данных обычно реализуют API уровня языка, такой как DB-API для Python или JDBC для Java.

Приложения, написанные в синхронном блокирующем стиле, часто используют тот же подход для доступа к данным. Кроме того, обычно взаимодействие с внешней базой данных через драйвер ничем не отличается от взаимодействия с внешней службой HTTP. Например, драйвер Apache Phoenix JDBC основан на компоненте Avatica из фреймворка Apache Calcite и использует обмен данными в формате JSON или Protocol Buffer через HTTP. То есть теоретически мы можем применить реактивный дизайн к протоколам связи с базами данных и получить схожие преимущества, как в случае с реактивным WebClient из модуля Spring WebFlux. Диаграмма на рис. 7.12 показывает, насколько похожи HTTP-запрос и запрос к базе данных с точки зрения сетевых взаимодействий.

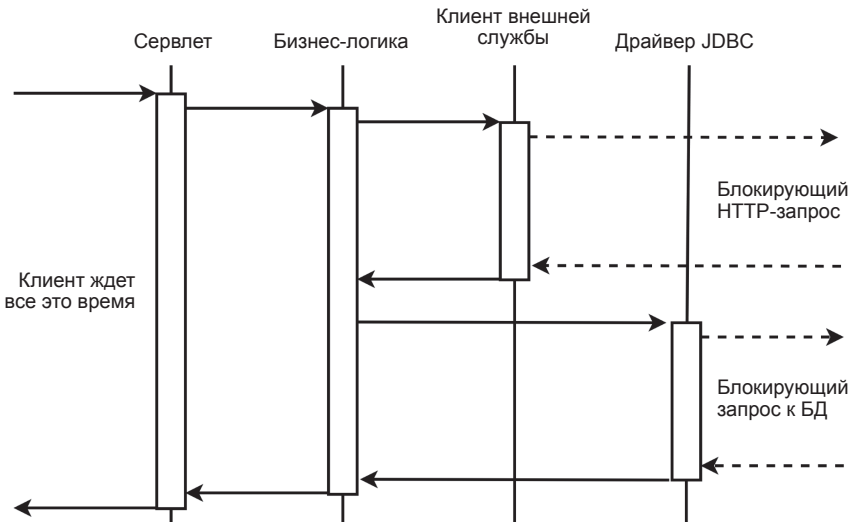


Рис. 7.12. Схожее поведение блокирующих HTTP-запроса и запроса к базе данных

Обычно блокирующая или неблокирующая природа драйверов баз данных определяется API верхнего уровня, а не протоколом связи. Поэтому не особенно сложно реализовать реактивный драйвер базы данных, который имеет соответствующий API уровня языка. Кандидаты на такой API рассматриваются далее в этой главе. В то же время драйверы баз данных NoSQL не имеют четко определенного API уровня языка для реализации, поэтому они могут реализовать свой асинхронный или реактивный API. По этому пути, например, решили пойти MongoDB, Cassandra и Couchbase, и теперь они предлагают либо асинхронные, либо реактивные драйверы.

JDBC

В 1997 году появился стандарт **Java Database Connectivity (JDBC)**, который с тех пор определяет, как приложения должны взаимодействовать с базами данных (главным образом реляционными), и предлагает унифицированный API для до-

ступа к данным на платформе Java. Последняя версия API (4.3) вышла в 2017 году и была включена в Java SE 9.

JDBC позволяет приложению иметь и одновременно использовать несколько драйверов для доступа к базе данных. За регистрацию, загрузку и использование необходимой реализации драйвера отвечает JDBC Driver Manager. Загрузив драйвер, клиент может создать соединение с соответствующими учетными данными. Соединения JDBC позволяют инициализировать и выполнять такие инструкции языка SQL, как SELECT, CREATE, INSERT, UPDATE и DELETE. Инструкции, которые в процессе выполнения изменяют состояние базы данных, возвращают число затронутых записей, а инструкции запроса возвращают экземпляр `java.sql.ResultSet`, поддерживающий итератор по записям в результате. Интерфейс `ResultSet` был разработан давно и имеет немного странный API. Например, нумерация столбцов в нем начинается с 1, а не с 0.

Интерфейс `ResultSet` предполагает возможность выполнения итераций в обратном порядке и даже произвольный доступ, но для этого необходимо, чтобы драйвер загрузил все записи до любого обращения к ним. Для простоты предположим, что `ResultSet` напоминает несложный итератор по записям в наборе данных. Это предположение позволяет базовой реализации оперировать набором результатов, разбитым на блоки, и загружать их из базы данных по требованию. Любая асинхронная базовая реализация должна быть заключена в синхронные блокирующие вызовы на уровне JDBC.

Для большей производительности JDBC поддерживает пакетную обработку запросов, не связанных с выборкой данных. Это должно ускорить взаимодействие с базой данных за счет уменьшения числа сетевых запросов. Однако это мало помогает в обработке больших наборов данных, потому что JDBC предполагает синхронную и блокирующую обработку.

Несмотря на то что JDBC проектировался как API уровня бизнес-логики, он работает с таблицами, записями и столбцами, а не с *сущностями* и *агрегатами*, как рекомендуется архитектурой DDD. По этой причине в настоящее время JDBC считается слишком низкоуровневым для непосредственного использования. Для таких целей в экосистеме Spring имеются модули Spring Data JDBC и Spring Data JPA. Существует также множество хорошо зарекомендовавших себя библиотек, обернутых JDBC и предлагающих более удобный API. Примером таких библиотек может служить Jdbi. Она не только предлагает текучий (fluent) API, но и имеет превосходную интеграцию с экосистемой Spring.

Управление соединениями

Современные приложения редко создают JDBC-соединения непосредственно. Чаще они используют **пул соединений**. Причина этого довольно проста – создание нового соединения обходится дорого. Поэтому разумнее иметь кеш соединений, которые можно использовать многократно. Цена создания соединений

складывается из двух слагаемых. Во-первых, процесс инициации соединения может требовать аутентификации и авторизации клиента, а на это уходит драгоценное время. Во-вторых, новое соединение может стоить базе данных очень дорого. Например, для каждого нового соединения PostgreSQL запускает новый отдельный процесс (не поток выполнения!), что может занимать сотни миллисекунд на не самой слабой машине с Linux. На момент написания этих строк наиболее широко на платформе Java использовались следующие пулы соединений: Apache Commons DBCP2, C3P0, Tomcat JDBC и HikariCP. Последний из них – HikariCP считается самым быстрым пулом соединений в мире Java.

Имейте в виду, что хотя пулы соединений широко используются для организации соединений JDBC, они не являются неотъемлемой частью реализации взаимодействий с базами данных. Например, драйвер базы данных Oracle допускает мультиплексирование соединений, что позволяет организовать несколько логических соединений, используя одно сетевое соединение. Конечно, такая поддержка обеспечивается не только драйвером, но и протоколом связи и самой реализацией базы данных.

Реактивный доступ к базе данных

JDBC считается основным API уровня языка для доступа к данным в мире Java (по крайней мере, когда речь идет о реляционных источниках данных), поэтому он влияет на поведение всех уровней абстракции, построенных на его основе. Выше мы узнали, почему блокирующие API нежелательно использовать в реактивных приложениях, – они ограничивают возможности масштабируемости. Поэтому нам крайне важно иметь API уровня языка для доступа к базе данных, который можно было бы использовать в реактивных приложениях. К сожалению, нет простых решений, которые могли бы немного подправить JDBC. На данный момент существует два многообещающих проекта API, которые могут заполнить эту нишу, и мы рассмотрим их далее в этой главе. На рис. 7.13 показано, что нужно, чтобы получить реактивный JDBC API.

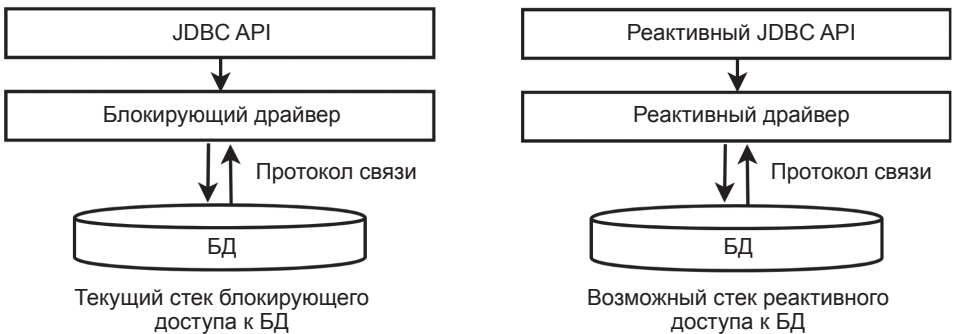


Рис. 7.13. Текущий стек JDBC и возможная реактивная замена

Spring JDBC

Для устранения неудобств, связанных с непосредственным использованием JDBC, в Spring имеется модуль Spring JDBC, довольно старый, но хорошо описанный. Этот модуль предлагает несколько версий класса `JdbcTemplate`, помогающего выполнять запросы и отображающего записи в сущности. Он также принимает на себя хлопоты, связанные с выделением и освобождением ресурсов, избавляя от распространенных ошибок, когда пользователь забывает закрыть параметризованный запрос или соединение. `JdbcTemplate` также перехватывает исключения JDBC и преобразует их в обобщенные исключения `org.springframework.dao`.

Допустим, у нас в базе данных SQL имеется коллекция книг, а в программе определена следующая сущность:

```
class Book {
    private int id;
    private String title;

    public Book() { }

    public Book(int id, String title) {
        this.id = id;
        this.title = title;
    }

    //методы свойств ...
}
```

Используя `JdbcTemplate` и шаблонный класс `BeanPropertyRowMapper`, можем создать репозиторий, как показано ниже.

```
@Repository
class BookJdbcRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public Book findById(int id) {
        return jdbcTemplate.queryForObject(
            "SELECT * FROM book WHERE id=?",
            new Object[] { id },
            new BeanPropertyRowMapper<>(Book.class));
    }
}
```

При желании можно определить свой класс, реализующий преобразование записей из `ResultSet` в предметные сущности.

```
class BookMapper implements RowMapper<Book> {  
    @Override  
    public Book mapRow(ResultSet rs, int rowNum) throws SQLException {  
        return new Book(rs.getInt("id"), rs.getString("title"));  
    }  
}
```

Давайте реализуем метод `BookJdbcRepository.findAll()`, используя класс `BookMapper`.

```
public List<Book> findAll() {  
    return jdbcTemplate.query("SELECT * FROM book", new BookMapper());  
}
```

Еще одно улучшение к `JdbcTemplate` предлагает класс `NamedParameterJdbcTemplate`. Он добавляет поддержку парсинга параметров JDBC с использованием удобочитаемых имен вместо знака вопроса (?). В результате параметризованный SQL-запрос и соответствующий код на Java могли бы выглядеть примерно так:

```
SELECT * FROM book WHERE title = :searchtitle
```

А вот как выглядит обычный параметризованный SQL-запрос:

```
SELECT * FROM book WHERE title = ?
```

Это улучшение может показаться незначительным, но именованные параметры улучшают читаемость кода, что особенно заметно, когда запрос принимает с полдесятка параметров.

Итак, модуль `Spring JDBC` включает множество утилит, вспомогательных классов и инструментов, обеспечивающих абстракции более высокого уровня. Высокоуровневый API не ограничен возможностями модуля `Spring JDBC` и позволяет относительно легко организовать поддержку реактивности, особенно если базовый API также поддерживает ее.

Spring Data JDBC

`Spring Data JDBC` – это относительно новый модуль в семействе `Spring Data`. Его цель – упростить реализацию хранилищ на основе JDBC. Хранилища `Spring Data`, включая основанные на JDBC, реализованы по образу и подобию хранилищ, описанных Эриком Эвансом (Eric Evans) в книге «*Domain-Driven Design*». Соответственно, мы рекомендуем создавать отдельное хранилище для каждого корневого агрегата. `Spring Data JDBC` поддерживает CRUD-операции для простых агрегатов, аннотации `@Query` и события жизненного цикла сущностей.



Будьте внимательны: *Spring Data JDBC* и *Spring JDBC* – это разные модули!

Чтобы воспользоваться модулем Spring Data JDBC, мы должны изменить сущность Book и применить аннотацию `org.springframework.data.annotation.Id` к полю `id`. Хранилище требует, чтобы каждая сущность имела уникальный идентификатор, поэтому сущность Book, пригодная для использования с хранилищем, должна выглядеть так, как показано ниже.

```
class Book {
    @Id
    private int id;
    private String title;

    // остальные части остались без изменений
}
```

Теперь определим интерфейс `BookRepository`, унаследовав его от `CrudRepository<Book, Integer>`.

```
@Repository
public interface BookSpringDataJdbcRepository
    extends CrudRepository<Book, Integer> { // (1)

    @Query("SELECT * FROM book WHERE LENGTH(title) = " + // (2)
           "(SELECT MAX(LENGTH(title)) FROM book)")
    List<Book> findByLongestTitle(); // (2.1)

    @Query("SELECT * FROM book WHERE LENGTH(title) = " +
           "(SELECT MIN(LENGTH(title)) FROM book)")
    Stream<Book> findByShortestTitle(); // (3)

    @Async // (4)
    @Query("SELECT * FROM book b " +
           "WHERE b.title = :title")
    CompletableFuture<Book> findBookByTitleAsync( // (4.1)
        @Param("title") String title);

    @Async // (5)
    @Query("SELECT * FROM book b " +
           "WHERE b.id > :fromId AND b.id < :toId")
    CompletableFuture<Stream<Book>> findBooksByIdBetweenAsync( // (5.1)
        @Param("fromId") Integer from,
        @Param("toId") Integer to);
}
```

Пояснения к коду.

1. Унаследовав `CrudRepository`, наше хранилище книг получает десяток методов, реализующих основные CRUD-операции, такие как `save(...)`, `saveAll(...)`, `findById(...)`, `deleteAll()`.

2. Регистрация нестандартного метода для поиска книг с самыми длинными названиями путем определения SQL-запроса в аннотации `@Query`. Однако, в отличие от Spring JDBC, здесь не выполняется никаких преобразований `ResultSet`. `JdbcTemplate` в данном случае не требуется, и нам нужно лишь написать интерфейс. Spring Framework сам сгенерирует реализацию, позаботившись обо всех ловушках. В качестве результата метод `findByLongestTitle` (2.1) возвращает контейнер `List`, соответственно, клиент разблокируется, только получив полный список с результатами.
3. Как вариант хранилище может вернуть поток `Stream` книг, поэтому, когда клиент вызывает метод `findByShortestTitle` (3.1), API может разрешить ему в зависимости от реализации обработать первый элемент еще до завершения обработки запроса в базе данных. Конечно, это возможно, только если базовая реализация и сама база данных поддерживают такой режим работы.
4. В методе `findBookByTitleAsync` (4.1) хранилище использует поддержку асинхронного выполнения из Spring Framework. Метод возвращает `CompletableFuture`, поэтому поток выполнения клиента не будет заблокирован в ожидании результатов. К сожалению, главный поток выполнения все еще будет блокироваться, что обусловлено особенностями реализации JDBC.
5. Также есть возможность объединить `CompletableFuture` и `Stream`, как это сделано в методе `findBooksByIdBetweenAsync` (5.1). При таком подходе поток выполнения клиента не должен блокироваться, пока не появятся первые результаты, после чего появляется возможность пакетной обработки результатов. К сожалению, для первой стадии соответствующий поток выполнения должен быть заблокирован, а впоследствии блокируется и поток выполнения клиента для извлечения следующей порции данных. Это лучшее, чего можно добиться с использованием JDBC без поддержки реактивности.

Чтобы информировать Spring о необходимости генерировать реализацию `BookRepository` с поддержкой Spring Data JDBC, нужно добавить следующую зависимость в приложение на основе Spring Boot.

```
compile('org.springframework.data:spring-data-jdbc:1.0.0.RELEASE')
```

Также в конфигурацию приложения нужно добавить аннотацию `@EnableJdbcRepositories`. За кадром *Spring Data JDBC* использует *Spring JDBC* и *NamedParameterJdbcTemplate*, которые мы обсудили выше.

Spring Data JDBC – это довольно маленький модуль, предлагающий удобные возможности для несложной реализации хранения данных в небольшом микросервисе. Впрочем, он преднамеренно сохраняется простым и не включает такие аспекты объектно-реляционного отображения (ORM), как кеширование, отложенная загрузка сущностей и поддержка сложных отношений между сущностями. Для этих целей в экосистеме Java используется отдельный стандарт – **Java Persistence API (JPA)**.

Добавление реактивности в Spring Data JDBC

Spring Data JDBC – это часть более крупного проекта Spring Data Relational. Spring Data JDBC использует JDBC – блокирующий API, не подходящий для использования в реактивных стеках. На момент написания этих строк команда Spring Data разрабатывала стандарт R2DBC, определяющий, как драйверы должны предоставлять реактивную и неблокирующую интеграцию с базами данных. На основе этого стандарта, как ожидается, будет создан модуль Spring Data R2DBC. Диаграмма на рис. 7.14 показывает теоретический реактивный стек Spring Data Relational.

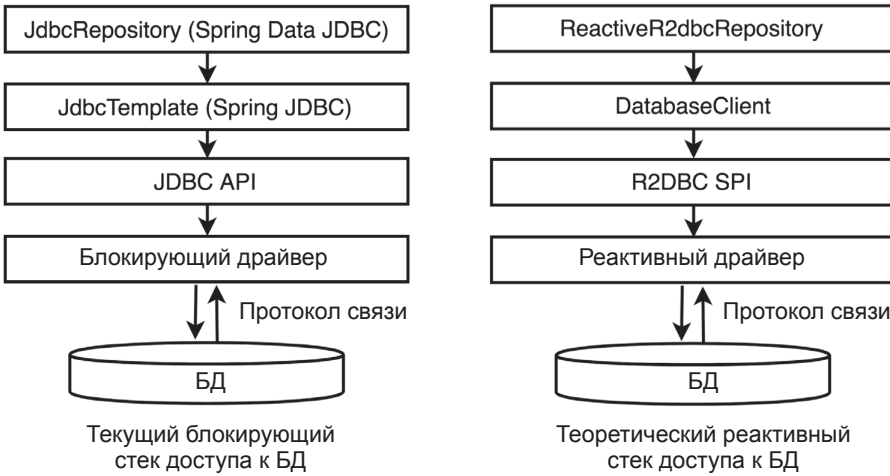


Рис. 7.14. Текущий стек Spring Data JDBC и возможная реактивная замена

JPA

Первая версия стандарта JPA появилась в 2006 году (последняя версия 2.2 была выпущена в 2013 году, иногда ее называют JPA2) и имела целью описать управление реляционными данными в приложениях на Java. В настоящее время стандарт JPA определяет организацию слоя хранения в приложениях. Он включает собственно определение API, а также описание языка запросов **Java Persistence Query Language (JPQL)**. JPQL – это SQL-подобный платформенно-независимый язык для извлечения объектов сущностей JPA из хранилищ.

В отличие от JDBC, стандарта, описывающего порядок доступа к базам данных, JPA – это стандарт **объектно-реляционного отображения** (Object Relational Mapping, ORM), который определяет детали отображения объектов в программном коде в таблицах в базе данных. Обычно ORM использует JDBC и динамически генерируемые SQL-запросы, но этот механизм по большей части скрыт от разработчиков. JPA позволяет отображать не только сущности, но и отношения между ними, упрощая загрузку связанных объектов.

Наибольшей популярностью пользуются две реализации JPA: Hibernate (<http://hibernate.org>) и EclipseLink (<http://www.eclipse.org/eclipselink>). Обе реализуют JPA 2.2 и являются взаимозаменяемыми. Кроме реализации стандарта JPA оба проекта предлагают дополнительный набор возможностей, отсутствующих в стандарте, но часто очень удобных. Например, EclipseLink позволяет обрабатывать события изменения базы данных и описывать отображение сущностей в таблицах в нескольких базах данных. Hibernate, в свою очередь, предлагает лучшую поддержку меток времени и естественных идентификаторов. Обе библиотеки поддерживают мультитенантность. Однако вы должны понимать, что при использовании уникальных возможностей библиотеки перестают быть взаимозаменяемыми.

Еще одна причина применения реализации JPA вместо чистой реализации JDBC – поддержка кеширования в Hibernate и EclipseLink. Обе библиотеки позволяют уменьшить число фактических обращений к базе данных за счет сохранения сущностей в кеше. Эта возможность способна оказывать существенное влияние на производительность приложения.

Добавление реактивности в JPA

На момент написания этих строк не было известно ничего о попытках придать асинхронность или реактивность реализациям JPA. Для этого необходимо иметь асинхронную или реактивную замену JDBC, на которой основан JPA. Вдобавок ко всему стандарт JPA разрабатывался с учетом многих допущений, неприменимых в реактивном программировании. Более того, огромная кодовая база JPA не самая легкая цель для реактивного рефакторинга. Проще говоря, в ближайшее время не стоит ждать появления поддержки реактивности в JPA.

Spring Data JPA

Spring Data JPA, как и Spring Data JDBC, тоже позволяет создавать хранилища, но внутри использует намного более мощную реализацию на основе JPA. Spring Data JPA обладает отличной поддержкой Hibernate и EclipseLink. Spring Data JPA может динамически генерировать JPA-запросы, опираясь на соглашения об именовании методов, предлагает реализацию шаблона Generic DAO (обобщенный объект доступа к данным) и добавляет поддержку библиотеки Querydsl (<http://www.querydsl.com>), помогающей писать компактные и типизированные запросы на Java.

Теперь напишем простейшее приложение, демонстрирующее основы Spring Data JPA. Следующая зависимость добавит все модули, необходимые приложению Spring Boot.

```
compile('org.springframework.boot:spring-boot-starter-data-jpa')
```

Spring Boot способна определить, что приложение использует Spring Data JPA, поэтому нет необходимости добавлять аннотацию `@EnableJpaRepositories` (хотя при желании это можно сделать). Определение сущности `Book` теперь должно выглядеть, как показано ниже.

```
@Entity
@Table(name = "book")
public class Book {
    @Id
    private int id;
    private String title;

    // Конструкторы, методы свойств...
}
```

Сущность `Book` отмечена аннотацией `javax.persistence.Entity`, позволяющей задать имя сущности, которое должно использоваться в JPQL-запросах. Аннотация `javax.persistence.Table` определяет таблицу и при необходимости ключи и индексы. Важно отметить, что вместо аннотации `org.springframework.data.annotation.Id` здесь используется аннотация `javax.persistence.Id`.

Теперь определим хранилище с нестандартным методом, имя которого выбрано в соответствии с соглашением об именовании, помогающим генерировать запросы, и с еще одним методом, использующим JPQL-запрос.

```
@Repository
interface BookJpaRepository
    extends CrudRepository<Book, Integer> {
    Iterable<Book> findByIdBetween(int lower, int upper);

    @Query("SELECT b FROM Book b WHERE LENGTH(b.title) = " +
        "(SELECT MIN(LENGTH(b2.title)) FROM Book b2)")
    Iterable<Book> findShortestTitle();
}
```

Драйвера JDBC, находящегося в пути поиска классов (classpath), одной зависимости Spring Boot, класса сущности `Book` и интерфейса `BookJpaRepository` достаточно, чтобы получить простейший, но довольно гибкий слой хранения данных, основанный на стеке технологий, включающем Spring Data JPA, JPA, JPQL, Hibernate и JDBC.

Добавление реактивности в Spring Data JPA

К сожалению, чтобы получить реактивный вариант модуля Spring Data JPA, необходимо добавить реактивность во все нижележащие уровни, включая JDBC, JPA и провайдера JPA. По этой причине появление реактивной версии в течение ближайших лет крайне маловероятно.

Spring Data NoSQL

Spring Data JPA и Spring Data JDBC – два замечательных решения для организации взаимодействий с реляционными базами данных, по крайней мере с теми, которые предлагают драйвер JDBC, но большинство баз данных NoSQL не имеет такого драйвера. Для подобных случаев в проекте Spring Data имеется несколько отдельных модулей, реализующих поддержку популярных баз данных NoSQL. Команда Spring активно разрабатывает модули для MongoDB, Redis, Apache Cassandra, Apache Solr, Gemfire, Geode и LDAP. Параллельно сообществом разрабатываются модули для следующих баз данных и хранилищ: Aerospike, ArangoDB, Couchbase, Azure Cosmos DB, DynamoDB, Elasticsearch, Neo4j, Google Cloud Spanner, Hazelcast и Vault.



Важно отметить, что обе библиотеки, EclipseLink и Hibernate, тоже поддерживают базы данных NoSQL. EclipseLink поддерживает MongoDB, Oracle NoSQL, Cassandra, Google BigTable и Couch DB. Подробности о поддержке баз данных NoSQL в EclipseLink можно найти в статье <https://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL>. В Hibernate имеется отдельный подпроект с названием Hibernate OGM (<http://hibernate.org/ogm>), направленный на поддержку баз данных NoSQL, таких как Infinispan, MongoDB, Neo4j и др. Однако из-за того что JPA по своей природе является реляционным API, в этих решениях, в отличие от специализированных модулей в Spring Data, отсутствуют возможности, характерные для NoSQL. Кроме того, использование JPA с его реляционной природой может увести в неверном направлении проектирование приложений, которые используют хранилища NoSQL.

Для работы с MongoDB можно использовать почти тот же код, что был показан в примере с Spring Data JDBC. Чтобы использовать хранилище MongoDB, нужно добавить следующую зависимость:

```
compile('org.springframework.boot:spring-boot-starter-data-mongodb')
```

Представим, что мы реализуем онлайн-каталог книг, и наше решение должно базироваться на MongoDB и Spring Framework. С этой целью определим сущность Book, как показано ниже.

```
@Document(collection = "book") // (1)
public class Book {
    @Id // (2)
    private ObjectId id; // (3)

    @Indexed // (4)
    private String title;

    @Indexed
    private List<String> authors; // (5)
```

```

    @Field("pubYear")                                // (6)
        private int publishingYear;

    // конструкторы, методы свойств
    // ...
}

```

Здесь вместо аннотации `@Entity` из JPA используется аннотация `@Document` (1) из пакета `org.springframework.data.mongodb.core.mapping`. Эта аннотация характерна для MongoDB и позволяет сослаться на нужную коллекцию в базе данных. Кроме того, чтобы определить внутренний идентификатор для сущности, мы использовали тип `org.bson.types.ObjectId` (3), также характерный для MongoDB, в сочетании с аннотацией `org.springframework.data.annotation.Id` (2) из Spring Data. Наша сущность, а значит, и документ в базе данных будут иметь поле `title` с названием, которое должно индексироваться базой данных MongoDB. С этой целью поле отмечено аннотацией `@Indexed` (4). Аннотация предлагает несколько параметров для настройки индексирования. Книга может иметь несколько авторов, поэтому мы определили поле `authors` с типом `List<String>` (5). Это поле тоже должно индексироваться. Обратите внимание, что здесь мы не определяем ссылки на отдельную таблицу `author` с отношением «многие ко многим», как это наверняка было бы реализовано в реляционной базе данных, а просто добавляем имена авторов как вложенный документ в сущность `Book`. Наконец, мы определили поле `publishingYear`. В сущности и в базе данных это поле имеет разные имена. Аннотация `@Field` позволяет настроить соответствие для таких случаев (6).

В базе данных такая сущность `Book` может быть представлена следующим документом JSON:

```

{
    "_id" : ObjectId("5b1c0908eb696eddfadc0b1b"),          /*(1)*/
    "title" : "The Expanse: Leviathan Wakes",
    "pubYear" : 2011,                                       /*(2)*/
    "authors" : [                                           /*(3)*/
        "Daniel Abraham",                                  /* */
        "Ty Franck"                                        /* */
    ],
    "_class" : "org.rpis5.chapters.chapter_07.mongo_repo.Book" /*(4)*/
}

```

Как видите, MongoDB использует особый тип данных для представления идентификатора документа (1). Поле `publishingYear` в данном случае отображается в `pubYear` (2), а поле `authors` представлено массивом (3). Кроме того, Spring Data MongoDB добавляет поддержку поля `_class`, описывающего Java-класс, который используется в объектно-документном отображении (Object-Document Mapping).

Интерфейс хранилища на основе MongoDB должен наследовать интерфейс `org.springframework.data.mongodb.repository.MongoRepository` (1), в свою очередь наследующий `CrudRepository`, который мы уже использовали в примерах выше.

```
@Repository
public interface BookSpringDataMongoRepository
    extends MongoRepository<Book, Integer> {                                // (1)

    Iterable<Book> findByAuthorsOrderByPublishingYearDesc(                // (2)
        String... authors
    );

    @Query("{ 'authors.1': { $exists: true } }")                            // (3)
    Iterable<Book> booksWithFewAuthors();

}
```

Конечно, хранилище на основе MongoDB поддерживает динамическое создание запросов, опираясь на соглашения об именовании, поэтому метод `findByAuthorsOrderByPublishingYearDesc` будет искать книги по именам авторов и возвращать результаты, отсортированные по году издания в обратном хронологическом порядке. Кроме того, аннотация `org.springframework.data.mongodb.repository.Query` позволяет писать запросы, характерные для MongoDB. Например, запрос (3) ищет книги, написанные несколькими авторами.

В остальном приложение должно работать точно так же, как при использовании Spring Data JDBC или Spring Data JPA.

Даже притом что мы рассмотрели все основные подходы к хранению данных в Spring, мы лишь слегка затронули эту область. Оставлены в стороне вопросы управления транзакциями, инициализации базы данных и миграции (Liquibase, Flyway), а также приемы отображения сущностей, кеширования и настройки производительности. Для описания всего этого может потребоваться не одна книга, но мы должны двигаться дальше и исследовать возможность реактивного доступа к хранимым данным.

Чтобы получить поддержку реактивности для баз данных NoSQL в Spring Framework, необходимо, чтобы вся базовая инфраструктура имела реактивный или асинхронный API. Вообще говоря, базы данных NoSQL появились относительно недавно и продолжают быстро развиваться, поэтому объем инфраструктуры, связанной с синхронным и блокирующим API, не так велик. Следовательно, добиться реактивности с базами данных NoSQL должно быть проще. В настоящий момент в Spring Data имеется несколько коннекторов для реактивной передачи данных и MongoDB в их числе. Мы поговорим об этом позже, в разделе «*Реактивный доступ к данным с использованием Spring Data*».

Ограничения синхронной модели

Исследуя механизмы хранения данных в Spring Framework и в Java в целом, мы рассмотрели JDBC, JPA, Hibernate, EclipseLink, Spring Data JDBC и Spring Data JDBС. Все эти API и библиотеки фактически действуют синхронно и блокируют выполнение клиента. Даже притом что почти все эти API извлекают данные, обращаясь к внешним службам, они не поддерживают неблокирующих взаимодействий. Соответственно, все перечисленные API противоречат парадигме реактивного программирования. Поток выполнения в Java, посылающий запрос в базу данных, неизбежно блокируется, пока не будет получена первая порция данных или не истечет тайм-аут, что выглядит расточительством с точки зрения управления ресурсами в реактивных приложениях. Как рассказывалось в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux», этот подход существенно ограничивает пропускную способность приложения и требует использовать гораздо более мощные серверы, а значит, затрачивать больше денег на их приобретение.

Выполнение запросов ввода/вывода в блокирующем режиме – неоправданное расточительство, будь то HTTP-запросы или запросы к базе данных. Кроме того, для взаимодействий с привлечением JDBC обычно используется целый пул соединений для параллельного выполнения запросов. В отличие от этого решения, широко используемый протокол HTTP2 позволяет задействовать одно TCP-соединение для одновременной работы с несколькими ресурсами. Этот подход способствует уменьшению количества занятых сокетов TCP и обеспечивает более высокую степень параллелизма и для клиентов, и для сервера (в данном случае это база данных). Взгляните на диаграмму на рис. 7.15.

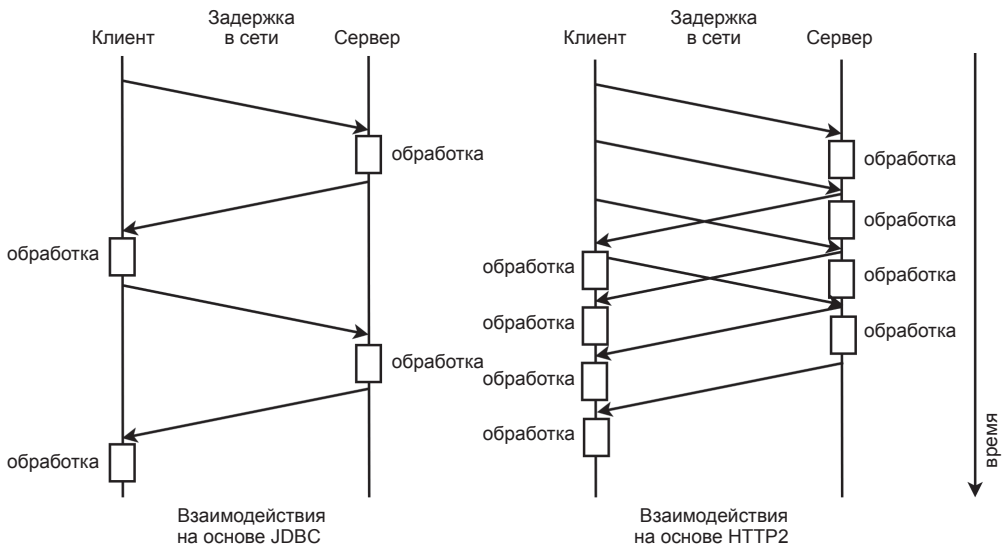


Рис. 7.15. Сравнение обычного протокола связи с базой данных и протокола с поддержкой мультиплексирования, такого как HTTP2

Да, пулы соединений экономят время, необходимое для открытия новых соединений. Также можно реализовать слой взаимодействий ниже уровня JDBC, чтобы организовать мультиплексирование, как в HTTP2, но код, предшествующий слою JDBC, все еще должен быть синхронным и блокирующим.

Аналогично при обработке объемных результатов запроса, занимающих несколько пакетов, взаимодействие с курсором базы данных (управляющая структура, поддерживающая итерации по записям в наборе результатов) выглядит так, как показано слева на рис. 7.15. В главе 3 «*Reactive Streams – новый стандарт потоков*» мы проанализировали различия между способами взаимодействий с точки зрения реактивных потоков, но все то же самое применимо к сетевым взаимодействиям.

Даже если база данных предложит асинхронный и неблокирующий драйвер, способный эффективно обмениваться данными и реализующий мультиплексирование соединений, мы все равно не сможем в полной мере использовать его потенциал с JDBC, JPA или Spring Data JPA. Следовательно, чтобы создать полностью реактивное приложение, мы должны отказаться от синхронных методик и создать API, использующий реактивные типы.

В заключение можно сказать, что традиционные и хорошо зарекомендовавшие себя реализации JDBC и JPA могут стать узким местом в современном реактивном приложении. JDBC и JPA почти наверняка будут использовать слишком много потоков выполнения и памяти и применения агрессивного кеширования, чтобы уменьшить количество длительных синхронных запросов и блокирующего ввода/вывода.

Нельзя сказать, что синхронная модель плоха, просто она плохо вписывается в реактивные приложения и почти всегда оказывается ограничивающим фактором. Однако эти модели могут успешно сосуществовать. И синхронный, и реактивный подходы имеют свои плюсы и минусы. Например, реактивный подход к хранению данных до сих пор не может предложить никаких решений для ORM, хотя бы чуточку приближающихся к JPA по своим возможностям.

Достоинства синхронной модели

Несмотря на то что синхронный доступ данных является не самым эффективным с точки зрения расходования ресурсов сервера, он остается высокоценным подходом, особенно при создании блокирующих веб-приложений. JDBC, будучи, пожалуй, самым популярным и универсальным API доступа к данным, почти идеально скрывает сложность клиент-серверных взаимодействий между приложением и базой данных. Spring Data JDBC и Spring Data JPA предлагают инструменты еще более высокого уровня, скрывая дополнительные сложности, связанные с трансляцией запросов и управлением транзакциями. Все эти проверенные временем решения значительно упрощают разработку современных приложений.

Синхронный доступ к данным прост в отладке и тестировании. При его использовании также легко контролировать расходование ресурсов, управляя пулом потоков выполнения. Синхронный подход предлагает широчайший спектр инструментов (таких как коннекторы JPA и Spring Data), не поддерживающих возможности управления обратным давлением, но все еще достаточно эффективных при использовании итераторов и синхронных потоков данных. Кроме того, большинство современных баз данных внутренне использует блокирующую модель, поэтому применение блокирующих драйверов для взаимодействий с ними выглядит естественным решением. Синхронный подход имеет превосходную поддержку локальных и распределенных транзакций. Также не составляет большого труда создать свою оболочку для драйвера, написанного на C или C++.

Единственный недостаток синхронного доступа к данным – блокирующий характер работы, который плохо совместим с реактивными приложениями, построенными с использованием реактивной парадигмы (Netty, Reactor, WebFlux).

Теперь, после краткого обзора синхронных способов доступа к данным, можем перейти к исследованию реактивных приемов хранения данных и посмотреть, как реактивные коннекторы в Spring Data помогают добиться высокой производительности без ущерба для универсальности хранилищ Spring Data.

Реактивный доступ к данным с использованием Spring Data

Итак, для создания полностью реактивного приложения нам нужно хранилище, работающее не с коллекциями, а с реактивными потоками сущностей. Реактивное хранилище должно позволять сохранять, изменять и удалять сущности, потребляя не только сами экземпляры Entity, но и реактивные Publisher<Entity>. Оно должно также возвращать данные в виде реактивных типов. В идеале хотелось бы иметь возможность работать с хранилищем способом, напоминающим WebClient из модуля Spring WebFlux. В действительности модуль Spring Data Commons предлагает интерфейс ReactiveCrudRepository с подобным контрактом.

Теперь обсудим, какие преимущества можно ожидать от использования реактивного доступа к данным вместо блокирующего. В главе 3 «*Reactive Streams – новый стандарт потоков*» мы уже сравнивали синхронную и реактивную модели извлечения данных, поэтому знаем, что при наличии идеального решения реактивного доступа к данным наше приложение может пользоваться всеми следующими преимуществами:

- **эффективное управление потоками выполнения**, потому что они никогда не блокируются в операциях ввода/вывода. Обычно это означает, что приложение может ограничиться меньшим количеством потоков выполнения, расходовать меньше вычислительных ресурсов на планирование пото-

ков и меньше памяти для размещения стеков объектов Thread и, следовательно, обрабатывать большее количество соединений;

- **меньший интервал времени от отправки запроса до получения первых результатов.** Первые результаты могут поступить еще до завершения обработки запроса. Это удобно для поисковых систем и интерактивных компонентов пользовательского интерфейса, нацеленных на работу с низкой задержкой;
- **меньшее потребление памяти.** При потоковой обработке требуется буферизовать меньше данных. Кроме того, клиент может отменить подписку на реактивный поток данных и уменьшить объем данных, передаваемых по Сети, как только получит объем, достаточный для удовлетворения его потребностей;
- **посредством обратного давления** клиент может информировать базу данных о готовности потребить новые данные. Аналогично сервер базы данных может информировать клиента о готовности обработать новые запросы. В этом случае есть возможность выполнить более срочную работу;
- еще одно преимущество может исходить из того факта, что реактивные клиенты не привязаны к потоку выполнения, поэтому отправка запроса и операции по обработке данных могут осуществляться в разных потоках выполнения. Базовые запросы и объекты соединений должны поддерживать такой режим работы. Так как ни один поток выполнения не обладает исключительными правами на объекты запросов и клиентский код никогда не блокируется, есть возможность **совместно использовать одно соединение с базой данных** и забыть о пугах соединений. Если база данных поддерживает режим интеллектуального соединения (smart connection mode), результаты выполнения запросов могут передаваться через единственное физическое сетевое соединение и направляться правильным реактивным подписчиком;
- и последнее в списке, но не последнее по важности. **Бесшовная интеграция слоя хранения данных с текучим (fluent) реактивным кодом** реактивного приложения подкреплена стандартом Reactive Streams.

Чем ближе к идеалу реактивный стек доступа к базе данных, тем больше преимуществ получит приложение. Однако некоторые из преимуществ, перечисленных выше, можно получить, используя асинхронный драйвер или даже блокирующий драйвер, завернутый в подходящий реактивный адаптер. Приложение может не иметь возможности оказывать обратное давление, но все еще может потреблять меньше памяти и эффективнее управлять потоками выполнения. Теперь пришло время поэкспериментировать с реактивным кодом в приложении Spring Boot.

Чтобы организовать поддержку реактивного хранения данных в приложении Spring Boot, нужно использовать одну из баз данных, для которых имеются реактивные коннекторы. На момент написания этих строк проект Spring Data предлагал поддержку реактивного подключения к MongoDB, Cassandra, Redis

и Couchbase. Этот список может показаться ограниченным, но на данный момент реактивность в технологиях хранения данных все еще остается новинкой. Кроме того, главным фактором, мешающим команде Spring расширить список баз данных с реактивной поддержкой, является отсутствие реактивных или асинхронных драйверов. А теперь рассмотрим особенности реализации реактивного CRUD-хранилища на примере MongoDB.

Реактивное хранилище на основе MongoDB

Чтобы получить реактивный доступ к данным в MongoDB, нужно добавить следующую зависимость в проект Gradle:

```
compile 'org.springframework.boot:spring-boot-starter-data-mongodb-reactive'
```

Представьте, что мы решили реорганизовать наше простое приложение с MongoDB из предыдущего раздела и сделать его реактивным. Можем оставить сущность Book как есть, без всяких изменений. Все аннотации объектно-документного отображения для MongoDB одинаковы для синхронного и реактивного модулей MongoDB. Но в реализации хранилища мы должны заменить обычные типы реактивными.

```
public interface ReactiveSpringDataMongoBookRepository
    extends ReactiveMongoRepository<Book, Integer> {           // (1)

    @Meta(maxScanDocuments = 3)                                // (2)
    Flux<Book> findByAuthorsOrderByPublishingYearDesc(         // (3)
        Flux<String> authors
    );

    @Query("{ 'authors.1': { $exists: true } }")                 // (4)
    Flux<Book> booksWithFewAuthors();

}
```

Итак, теперь наше хранилище расширяет интерфейс ReactiveMongoRepository (1) вместо MongoRepository. В свою очередь, ReactiveMongoRepository расширяет интерфейс ReactiveCrudRepository, общий для всех реактивных коннекторов.



Несмотря на отсутствие RxJava2MongoRepository, мы все еще можем использовать реактивные хранилища из Spring Data с библиотекой RxJava 2, просто расширяя RxJava2CrudRepository. Модуль Spring Data поддерживает преобразование типов из Project Reactor в типы RxJava 2 и обратно, что позволяет нам использовать опыт работы с RxJava 2.

Интерфейс ReactiveCrudRepository – это реактивный эквивалент интерфейса CrudRepository из синхронного модуля Spring Data. Модуль Reactive Spring Data Repositories предлагает те же аннотации и аналоги большинства синхрон-

ных функций. Поэтому реактивное хранилище Mongo поддерживает конструирование запросов с использованием соглашений об именовании методов (3), аннотацию `@Query` с запросами к MongoDB, написанными вручную (4), и аннотацию `@Meta` с теми же дополнительными возможностями настройки запросов (2). Также поддерживаются конструкции для запуска **запросов по образцу** (Query by Example, QBE). Однако, в отличие от синхронного `MongoRepository`, реактивный `ReactiveMongoRepository` расширяет интерфейс `ReactiveSortingRepository`, позволяющий запрашивать результаты в определенном порядке, но не поддерживающий возможность постраничного получения результатов. Подробнее о проблеме постраничного получения данных мы поговорим в разделе «Поддержка разбиения на страницы».

Как обычно, мы можем внедрить в наше приложение компонент типа `ReactiveSpringDataMongoBookRepository`, и Spring Data предоставит нужную реализацию. Следующий пример показывает, как добавить несколько книг в MongoDB, используя реактивный подход.

```
@Autowired
private ReactiveSpringDataMongoBookRepository rxBookRepository; // (1)
...
Flux<Book> books = Flux.just(                                // (2)
    new Book("The Martian", 2011, "Andy Weir"),
    new Book("Blue Mars", 1996, "Kim Stanley Robinson")
);

rxBookRepository
    .saveAll(books)                                           // (3)
    .then()                                                  // (4)
    .doOnSuccess(ignore -> log.info("Books saved in DB"))    // (5)
    .subscribe();                                           // (6)
```

Пояснения к коду.

1. Внедряется компонент с интерфейсом `BookSpringDataMongoRxRepository`.
2. Подготавливается поток сущностей `Book` для добавления в базу данных.
3. Сущности сохраняются вызовом метода `saveAll`, который использует `Publisher<Book>`. Как обычно, фактическое сохранение не выполняется до появления подписчика. `ReactiveCrudRepository` имеет также версию метода `saveAll`, который использует интерфейс `Iterable`. Эти два метода реализуют разную семантику, но мы обсудим данный вопрос позже.
4. Метод `saveAll` возвращает `Flux<Book>` с сохраненными сущностями, но, так как этот уровень детализации нас не интересует, вызываем метод `then`, который преобразует поток так, что он вернет только одно из двух событий: `onComplete` или `onError`.
5. Когда поток выполнит свою работу и все книги будут записаны, мы выводим соответствующее сообщение в журнал.

6. Как обычно, всякий реактивный поток данных должен иметь подписчика. Здесь ради простоты примера мы подписываемся без использования каких-либо обработчиков. Однако в реальном приложении должны использоваться настоящие подписчики, которые обработают ответ.

Теперь извлечем данные из MongoDB, используя реактивный поток. Чтобы вывести результаты, поставляемые в реактивном потоке, можно использовать следующий вспомогательный метод:

```
private void reportResults(String message, Flux<Book> books) { // (1)
    books
        .map(Book::toString) // (2)
        .reduce( // (3)
            new StringBuilder(), // (3.1)
            (sb, b) -> sb.append(" - ") // (3.2)
                .append(b)
                .append("\n"))
        .doOnNext(sb -> log.info(message + "\n{", sb)) // (4)
        .subscribe(); // (5)
}
```

Рассмотрим подробно, что делает этот код:

1. Метод, который выводит удобочитаемый список книг в виде одного сообщения с указанным префиксом `message`.
2. Для каждой книги `book` в потоке вызывается ее метод `toString`, и дальше передается ее строковое представление.
3. Метод `Flux.reduce` собирает строковые представления всех книг в одно сообщение. Обратите внимание, что такое решение может оказаться неприемлемым для большого количества книг, потому что добавление каждой новой книги увеличивает размер буфера и может привести к чрезмерному потреблению памяти. Для сохранения промежуточных результатов мы используем класс `StringBuilder` (3.1). Имейте в виду, что `StringBuilder` не является безопасным в многопоточном окружении, а метод `onNext` может вызываться из разных потоков выполнения, но стандарт `Reactive Streams` гарантирует семантику `happens-before`. То есть даже если сущности будут вставляться разными потоками выполнения, их можно безопасно объединить с помощью `StringBuilder`, потому что барьеры доступа к памяти гарантируют последнее состояние объекта `StringBuilder` при обновлении внутри одного реактивного потока. В строке (3.2) строковое представление книги добавляется в конец буфера.
4. Метод `reduce` пошлет событие `onNext` только после обработки всех входящих событий `onNext`, поэтому мы без опаски можем вывести окончательное сообщение со всеми книгами.

5. Чтобы запустить обработку, нужно оформить подписку. Для простоты здесь предполагается невозможность ошибок. Однако в промышленном коде необходимо предусмотреть некоторую логику обработки ошибок.

Теперь прочитаем список книг в базе данных.

```
Flux<Book> allBooks = rxBookRepository.findAll();  
reportResults("All books in DB:", allBooks);
```

Следующий код отыщет все книги Энди Вейра (Andy Weir), используя соглашение об именовании методов.

```
Flux<Book> andyWeirBooks = rxBookRepository  
    .findByAuthorsOrderByPublishingYearDesc(Mono.just("Andy Weir"));  
reportResults("All books by Andy Weir:", andyWeirBooks);
```

Кроме того, передает критерии поиска с использованием типа `Mono<String>` и фактически посылает запрос в базу данных, только когда `Mono` сгенерирует событие `onNext`. Как видите, реактивное хранилище становится естественным продолжением реактивного потока, где входящий и исходящий потоки являются реактивными.

Объединение операций с хранилищем

Теперь реализуем чуть более сложный бизнес-сценарий. Допустим, нам нужно изменить год публикации по названию книги. То есть сначала нужно найти экземпляр книги, затем изменить год ее публикации и сохранить обновленную информацию в базе данных. Чтобы усложнить задачу, предположим также, что название и год публикации извлекаются асинхронно, с некоторой задержкой, и доставляются с помощью типа `Mono`. Также нам нужно знать, завершился ли запрос на изменение успешно. В данный момент мы не требуем, чтобы изменение выполнялось атомарно, и предполагаем, что в базе данных хранится информация только об одной книге с таким названием. Итак, с учетом всех перечисленных требований можем определить бизнес-метод с таким API.

```
public Mono<Book> updatedBookYearByTitle(                               // (1)  
    Mono<String> title,                                                // (2)  
    Mono<Integer> newPublishingYear)                                   // (3)
```

Пояснения к коду.

1. Метод `updatedBookYearByTitle` возвращает сущность книги для изменения (или ничего, если книга не найдена).
2. Значение `title` поступает как тип `Mono<String>`.
3. Новый год публикации поступает как тип `Mono<Integer>`.

Теперь можно написать тест, проверяющий работу нашего метода `updatedBookYearByTitle`.

```
Instant start = now(); // (1)
Mono<String> title = Mono.delay(Duration.ofSeconds(1)) // (2)
    .thenReturn("Artemis") //
    .doOnSubscribe(s -> log.info("Subscribed for title")) //
    .doOnNext(t -> //
        log.info("Book title resolved: {}", t)); // (2.1)

Mono<Integer> publishingYear = Mono.delay(Duration.ofSeconds(2)) // (3)
    .thenReturn(2017) //
    .doOnSubscribe(s -> log.info("Subscribed for publishing year")) //
    .doOnNext(t -> //
        log.info("New publishing year resolved: {}", t)); // (3.1)

updatedBookYearByTitle(title, publishingYear) // (4)
    .doOnNext(b ->
        log.info("Publishing year updated for book: {}", b)) // (4.1)
    .hasElement() // (4.2)
    .doOnSuccess(status ->
        log.info("Updated finished {}, took: {}", // (5)
            status ? "successfully" : "unsuccessfully",
            between(start, now()))) // (5.1)
    .subscribe(); // (6)
```

Пояснения к коду.

1. Используется для определения времени выполнения, хранит время запуска теста.
2. Имитирует получение названия с задержкой в одну секунду, выводит запись в журнал сразу после получения названия (2.1).
3. Имитирует получение нового года издания с задержкой в две секунды, выводит запись в журнал сразу после получения года издания (3.1).
4. Вызов бизнес-метода, выводит запись в журнал после получения подтверждения, если оно вообще будет получено (4.1). Для определения присутствия события `onNext` (означающего фактическое изменение информации о книге) вызывается метод `Mono.hasElement`, возвращающий `Mono<Boolean>`.
5. Когда поток данных завершится, выводит запись в журнал об успехе или неудаче операции обновления с общим временем выполнения.
6. Как обычно, чтобы реактивный конвейер запустился, нужно оформить подписку.

Из этого кода можно сделать вывод, что конвейер не сможет выполнить свою работу быстрее, чем за две секунды, потому что именно столько требуется для по-

лучения года издания. Но он может работать дольше. Давайте выполним первую итерацию в создании реализации.

```
private Mono<Book> updatedBookYearByTitle(      /* Первая итерация */
    Mono<String> title,
    Mono<Integer> newPublishingYear
) {
    return rxBookRepository.findOneByTitle(title)           // (1)
        .flatMap(book -> newPublishingYear                // (2)
        .flatMap(year -> {                                 // (3)
            book.setPublishingYear(year);                   // (4)
            return rxBookRepository.save(book);             // (5)
        }));
}
```

В самом начале метода мы обращаемся к хранилищу, используя реактивную ссылку на название книги (1). Как только сущность Book будет найдена (2), подписываемся на получение нового года издания. Затем, получив год издания, изменяем сущность Book (4) и вызываем метод save хранилища. У себя мы получили следующий вывод:

```
Subscribed for title
Book title resolved: Artemis
Subscribed for publishing year
New publishing year resolved: 2017
Publishing year updated for book: Book(publishingYear=2017...
Updated finished successfully, took: PT3.027S
```

Итак, информация о книге была успешно изменена, но, как можно видеть по записям в журнале, подписка на получение нового года издания была выполнена только после получения названия, поэтому в общей сложности метод затратил на работу чуть больше трех секунд. Мы можем ускорить процесс, подписавшись сразу на оба потока данных в самом начале, чтобы запустить два параллельных процесса извлечения данных. Следующий фрагмент показывает, как это можно реализовать с помощью метода zip.

```
private Mono<Book> updatedBookYearByTitle(      /* Вторая итерация */
    Mono<String> title,
    Mono<Integer> newPublishingYear
) {
    return Mono.zip(title, newPublishingYear)           // (1)
        .flatMap((Tuple2<String, Integer> data) -> {    // (2)
            String titleVal = data.getT1();              // (2.1)
            Integer yearVal = data.getT2();              // (2.2)
            return rxBookRepository
                .findOneByTitle(Mono.just(titleVal))     // (3)
                .flatMap(book -> {
```

```

        book.setPublishingYear(yearVal);           // (3.1)
        return rxBookRepository.save(book);        // (3.2)
    });
}

```

Здесь, вызовом `zip`, мы объединяем два значения и одновременно подписываемся на них (1). Как только оба значения будут получены, наш поток данных получит контейнер `Tuple2<String, Integer>` с интересующими нас значениями (2). Далее нужно распаковать значения (2.1) и (2.2) вызовами `data.getT1()` и `data.getT2()`. В строке (3) мы запрашиваем сущность `Book` и после ее получения изменяем год издания и сохраняем сущность в базе данных. После второй итерации разработки у нас это приложение вывело следующие строки:

```

Subscribed for title
Subscribed for publishing year
Book title resolved: Artemis
New publishing year resolved: 2017
Publishing year updated for the book: Book(publishingYear=2017...
Updated finished successfully, took: PT2.032S

```

На этот раз, как можете видеть сами, мы сначала подписываемся на оба потока данных сразу, а когда запрошенные значения поступают – изменяем сущность `Book`. Второе решение выполняется примерно две секунды вместо трех. Этот вариант работает быстрее, но требует использовать тип `Tuple2`, что вынуждает писать дополнительные строки кода и выполнять преобразования. Чтобы улучшить читаемость кода и убрать вызовы `getT1()` и `getT2()`, можно добавить модуль *Reactor Addons*, предлагающий некоторый синтаксический сахар для таких случаев.

Добавим следующую зависимость:

```
compile('io.projectreactor.addons:reactor-extra')
```

Таким образом улучшается предыдущий пример кода.

```

private Mono<Book> updatedBookYearByTitle(          /* Третья итерация */
    Mono<String> title,
    Mono<Integer> newPublishingYear
) {
    return Mono.zip(title, newPublishingYear)
        .flatMap(
            TupleUtils.function((titleValue, yearValue) ->           // (1)
                rxBookRepository
                    .findOneByTitle(Mono.just(titleValue))           // (2)
                    .flatMap(book -> {
                        book.setPublishingYear(yearValue);
                        return rxBookRepository.save(book);
                    })
            )
        )
    }

```

```

    }));
}

```

В строке (1) мы заменили извлечение элементов из объекта `Tuple2` вызовом метода `function` класса `TupleUtils` и продолжаем работать с уже извлеченными элементами. Так как метод `function` является статическим, мы получаем простой и понятный код.

```

return Mono.zip(title, newPublishingYear)
    .flatMap(function((titleValue, yearValue) -> { ... }));

```

В строке (2) вновь заворачиваем `titleValue` в объект `Mono`. Мы могли бы использовать исходный объект `title`, который уже имеет нужный тип, но тогда мы дважды подписались бы на поток `title` и могли бы получить следующий вывод. Обратите внимание, что код получения `title` выполняется дважды.

```

Subscribed for title
Subscribed for publishing year
Book title resolved: Artemis
New publishing year resolved: 2017
Subscribed for title
Book title resolved: Artemis
Publishing year updated for the book: Book(publishingYear=2017...
Updated finished successfully, took: PT3.029S

```

Важно также отметить, что в третьей итерации запускаем запрос на загрузку книги из базы данных только после получения обоих значений – названия и года издания. Однако загрузку сущности книги можно было бы начать раньше, когда запрос на извлечение года издания еще обрабатывался, но название книги уже получено. В четвертой итерации реализуем такой реактивный конвейер.

```

private Mono<Book> updatedBookYearByTitle( /* Четвертая итерация */
    Mono<String> title,
    Mono<Integer> newPublishingYear
) {
    return Mono.zip(
        newPublishingYear, // (1)
        rxBookRepository.findOneByTitle(title) // (1.1)
    ).flatMap(function((yearValue, bookValue) -> { // (1.2)
        bookValue.setPublishingYear(yearValue); // (2)
        return rxBookRepository.save(bookValue); // (2.1)
    }));
}

```

С помощью оператора `zip` (1) мы одновременно подписываемся на получение названия (1.2) и года издания (1.1). По прибытии обоих значений (2) изменяем год издания в сущности и посылаем запрос на сохранение (2.1). Кроме того, как

и во всех предыдущих итерациях, даже если никакие части конвейера не блокируются, для выполнения работы требуется не менее двух секунд. То есть этот код использует вычислительные ресурсы очень эффективно.

Главная цель упражнения – показать, что благодаря возможностям реактивных потоков и гибкости Project Reactor API можно легко конструировать разные асинхронные рабочие процессы, даже включающие слой хранения данных. С помощью всего нескольких реактивных операторов мы можем полностью изменить порядок передачи данных через систему. Однако не все реактивные альтернативы равны. Некоторые могут выполняться быстрее, другие медленнее, и во многих случаях самое очевидное решение оказывается не самым лучшим. Поэтому, разрабатывая реактивные конвейеры, обязательно рассматривайте альтернативные комбинации реактивных операторов и не останавливайтесь на первом найденном решении, а постарайтесь найти лучшее решение.

Как работают реактивные хранилища

Теперь рассмотрим детали работы реактивных хранилищ. Реактивное хранилище в Spring Data работает путем адаптации возможностей драйвера базы данных. На нижнем уровне может быть драйвер, асинхронный или совместимый со стандартом Reactive Streams, который можно завернуть в реактивный API. Здесь мы посмотрим, как реактивное хранилище MongoDB использует реактивный драйвер MongoDB и как реактивное хранилище Cassandra использует асинхронный драйвер.

Прежде всего отметим, что интерфейс `ReactiveMongoRepository` расширяет более общие интерфейсы `ReactiveSortingRepository` и `ReactiveQueryByExampleExecutor`. Интерфейс `ReactiveQueryByExampleExecutor` позволяет выполнять запросы QBE. Интерфейс `ReactiveSortingRepository` расширяет более общий интерфейс `ReactiveCrudRepository` и добавляет метод `findAll`, который позволяет указывать порядок сортировки результатов.

Поскольку многие коннекторы используют интерфейс `ReactiveCrudRepository`, рассмотрим его поближе. `ReactiveCrudRepository` объявляет методы для сохранения, поиска и удаления сущностей. Метод `Mono<T> save(T entity)` сохраняет `entity` и возвращает сохраненную сущность для дальнейших операций. Обратите внимание, что операция сохранения может полностью заменить объект сущности. Операция `Mono<T> findById(ID id)` получает идентификатор `id` сущности и возвращает результаты, завернутые в `Mono`. Метод `findAllById` имеет две версии, одна из них принимает идентификаторы в форме коллекции `Iterable<ID>`, а другая – в форме `Publisher<ID>`. Единственное заметное отличие между `ReactiveCrudRepository` и `CrudRepository` кроме поддержки реактивности – отсутствие в `ReactiveCrudRepository` поддержки транзакций и разбиения результатов на страницы. О поддержке транзакций в реактивных хранилищах с по-

мощью Spring Data мы поговорим далее в этой главе, а сейчас рассмотрим проблемы реализации разбиения на страницы, решение которых полностью лежит на плечах разработчика.

Поддержка разбиения на страницы

Важно отметить, что команда Spring Data намеренно исключила поддержку разбиения на страницы, потому что реализация в синхронных хранилищах не соответствует реактивной парадигме. Для вычисления параметров следующей страницы нужно знать число записей, полученных в предыдущем результате. Также, чтобы узнать общее число страниц, нужно знать общее число записей. Обе характеристики не вписываются в реактивную и неблокирующую парадигму. Кроме того, запросы к базе данных для подсчета всех записей обходятся довольно дорого и увеличивают задержку перед началом обработки первых результатов. И все же есть возможность извлекать данные порциями, передавая в хранилище объект `Pageable`, как показано ниже.

```
public interface ReactiveBookRepository
    extends ReactiveSortingRepository<Book, Long> {
    Flux<Book> findByAuthor(String author, Pageable pageable);
}
```

То есть теперь мы можем запросить вторую страницу с результатами (обратите внимание, что нумерация индексов начинается с 0), где каждая страница содержит пять элементов.

```
Flux<Book> result = reactiveBookRepository
    .findByAuthor('Andy Weir', PageRequest.of(1, 5));
```

Детали реализации ReactiveMongoRepository

Модуль MongoDB Reactive в Spring Data имеет только одну реализацию интерфейса `ReactiveMongoRepository` – класс `SimpleReactiveMongoRepository`. Он реализует все методы `ReactiveMongoRepository` и использует интерфейс `ReactiveMongoOperations` для поддержки всех низкоуровневых операций.

Взгляните на реализацию метода `findAllById(Publisher<ID> ids)`.

```
public Flux<T> findAllById(Publisher<ID> ids) {
    return Flux.from(ids).buffer().flatMap(this::findAllById);
}
```

Как видите, этот метод собирает все идентификаторы с помощью операции `buffer` и затем выполняет единственный запрос, вызывая версию `findAllById(Iterable<ID> ids)`. Эта версия, в свою очередь, формирует объект `Query`

и вызывает метод `findAll(Query query)`, который обращается к экземпляру `ReactiveMongoOperations`, `mongoOperations.find(query, ...)`.

Еще одно интересное наблюдение: метод `insert(Iterable<S> entities)` вставляет сущности `entities` в один пакетный запрос. В то же время метод `insert(Publisher<S> entities)` генерирует множество запросов внутри оператора `flatMap`, как показано ниже.

```
public <S extends T> Flux<S> insert(Publisher<S> entities) {  
    return Flux.from(entities)  
        .flatMap(entity -> mongoOperations.insert(entity, ...));  
}
```

В этом случае две версии метода `findById` действуют одинаково и генерируют только один запрос к базе данных. Теперь обратим внимание на метод `saveAll`. Версия метода, которая принимает `Publisher`, выполняет один запрос для каждой сущности. Версия, которая принимает `Iterable`, выполняет один запрос, когда все сущности являются новыми, и по одному запросу для каждой сущности в остальных случаях. Метод `deleteAll(Iterable<? extends T> entities)` всегда выполняет отдельный запрос для каждой сущности, даже если все сущности доступны в контейнере `Iterable` и нет необходимости ждать их получения.

Как видите, разные версии одного и того же метода могут действовать по-разному и генерировать разное число запросов. Кроме того, такое поведение слабо связано с получением входных значений из синхронного итератора или реактивного `Publisher`. Поэтому, чтобы узнать, сколько запросов к базе данных сгенерирует тот или иной метод, мы советуем заглянуть в его реализацию.

С методами `ReactiveCrudRepository`, реализации которых генерируются на лету, дело обстоит сложнее. Но и в этом случае генерация запросов выполняется аналогично обычному синхронному `CrudRepository`. `RepositoryFactorySupport` создает соответствующий прокси для `ReactiveCrudRepository`. Если метод декорирован аннотацией `@Query`, для создания запросов используется класс `ReactiveStringBasedMongoQuery`. Если запрос генерируется на основе соглашений об именовании методов, используется класс `ReactivePartTreeMongoQuery`. Кроме того, установив уровень журналирования `DEBUG` для `ReactiveMongoTemplate`, можно отследить все запросы, посылаемые в `MongoDB`.

Использование ReactiveMongoTemplate

Даже притом что `ReactiveMongoTemplate` служит строительным блоком в реализации реактивного хранилища, сам этот класс очень универсален. Иногда он позволяет увеличить эффективность взаимодействий с базой данных даже больше, чем высокоуровневая реализация хранилища.

Например, реализуем простую службу, которая использует `ReactiveMongoTemplate` для поиска книг по названию с помощью регулярного выражения.


```
public class RxMongoTemplateQueryService {  
    private final ReactiveMongoOperations mongoOperations;           // (1)  
    // Конструктор...  
  
    public Flux<Book> findBooksByTitle(String titleRegExp) {        // (2)  
        Query query = Query.query(new Criteria("title")           // (3)  
            .regex(titleRegExp)))  
            .limit(100);  
        return mongoOperations  
            .find(query, Book.class, "book");                       // (4)  
    }  
}
```

Вот наиболее примечательные аспекты класса `RxMongoTemplateQueryService`.

1. Нам нужна ссылка на экземпляр интерфейса `ReactiveMongoOperations`.
2. Класс `ReactiveMongoTemplate` реализует этот интерфейс и присутствует в контексте Spring, если настроен источник данных MongoDB.
3. Служба определяет метод `findBooksByTitle`, принимающий регулярные выражения как критерии поиска и возвращающий поток данных `Flux` с результатами.
4. Для формирования фактических запросов на основе регулярных выражений используются классы `Query` и `Criteria` коннектора MongoDB. Мы также ограничиваем число результатов значением 100, применив метод `Query.limit`.
5. Здесь обращаемся к `mongoOperations` для выполнения сконструированного запроса. Результаты запроса необходимо отобразить в сущности класса `Book`. Также мы должны указать имя коллекции для поиска. В предыдущем примере посылаем запрос на поиск в коллекции с названием `book`.



Обратите внимание, что того же поведения можно достичь (кроме ограничения числа результатов) с использованием обычного реактивного хранилища, определив метод с сигнатурой, следующей соглашению об именовании.

```
Flux<Book> findManyByTitleRegex(String regex).
```

За кадром `ReactiveMongoTemplate` использует интерфейс `ReactiveMongoDatabaseFactory` для получения экземпляра реактивного соединения с MongoDB. Кроме того, он использует экземпляр интерфейса `MongoConverter` для преобразования сущностей в документы и обратно. `MongoConverter` также используется в синхронном `MongoTemplate`. Давайте посмотрим, как `ReactiveMongoTemplate` реализует этот контракт. Например, метод `find(Query query, ...)` отображает экземпляр `org.springframework.data.mongodb.core.query.Query` в экземпляр класса `org.bson.Document`, с которым может работать клиент MongoDB. Затем `ReactiveMongoTemplate` вызывает клиента базы данных и передает ему преобразованный запрос. Класс `com.mongodb.reactivestreams.client.MongoClient`

предоставляет точку входа для реактивного драйвера MongoDB. Он совместим со стандартом Reactive Streams и возвращает данные посредством реактивных издателей.

Использование реактивных драйверов (MongoDB)

Реактивное подключение к MongoDB в Spring Data построено на основе реактивного драйвера MongoDB Reactive Streams Java Driver (<https://github.com/mongodb/mongo-java-driver-reactivestreams>). Драйвер поддерживает асинхронную потоковую обработку с неблокирующим обратным давлением. Сам драйвер, в свою очередь, базируется на MongoDB Async Java Driver (<http://mongodb.github.io/mongo-java-driver/3.8/driver-async>). Асинхронный драйвер имеет низкоуровневую реализацию и предлагает API на основе обратных вызовов, поэтому его не просто использовать в роли высокоуровневого драйвера с поддержкой реактивных потоков. Следует отметить, что помимо MongoDB Reactive Streams Java Driver существует также MongoDB RxJava Driver (<http://mongodb.github.io/mongo-java-driver-rx>), который основан на том же асинхронном драйвере MongoDB. Таким образом, для подключения к MongoDB в экосистеме Java есть один синхронный и два реактивных драйвера.

Конечно, если потребуется организовать более тонкое управление обработкой запросов, чем позволяет ReactiveMongoTemplate, можно использовать реактивный драйвер непосредственно. В такой ситуации предыдущий пример после перехода на непосредственное использование реактивного драйвера мог бы выглядеть примерно так:

```
public class RxMongoDriverQueryService {  
    private final MongoClient mongoClient; // (1)  
  
    public Flux<Book> findBooksByTitleRegex(String regex) { // (2)  
        return Flux.defer(() -> { // (3)  
            Bson query = Filters.regex(titleRegex); // (3.1)  
            return mongoClient //  
                .getDatabase("test-database") // (3.2)  
                .getCollection("book") // (3.3)  
                .find(query); // (3.4)  
        })  
        .map(doc -> new Book( // (4)  
            doc.getObjectId("id"),  
            doc.getString("title"),  
            doc.getInteger("pubYear"),  
            // ... другие процедуры отображения  
        ));  
    }  
}
```

Пояснения к коду.

1. Служба ссылается на экземпляр интерфейса `com.mongodb.reactivestreams.client.MongoClient`. Этот экземпляр должен быть доступен в виде компонента после настройки источника данных.
2. Определяется метод `findBooksByTitleRegex`, который возвращает поток Flux с сущностями Book.
3. Мы должны вернуть новый экземпляр Flux, который откладывает выполнение до момента появления фактического подписчика. Внутри этого лямбда-выражения определяем новый запрос с типом `org.bson.conversions.Bson`, используя вспомогательный класс `com.mongodb.client.model.Filters`. Затем ссылаемся на базу данных (3.2) и коллекцию (3.3) по именам. Никакого обмена с базой данных в этот момент не происходит, пока мы не отправим прежде подготовленный запрос (3.4) вызовом метода `find`.
4. Как только начнут поступать результаты, можем преобразовать документы MongoDB в предметные сущности, если это необходимо.

Опустившись в предыдущем примере до уровня драйвера базы данных, мы все еще чувствуем себя вполне комфортно, как если бы работали с реактивными потоками данных. Кроме того, нам не нужно обслуживать обратное давление вручную, потому что MongoDB Reactive Streams Java Driver уже поддерживает это. Реактивное подключение к MongoDB организует обратное давление исходя из размера пакета. Этот подход вполне приемлем как решение по умолчанию, но может генерировать значительный трафик при небольших увеличениях спроса. Диаграмма на рис. 7.16 перечисляет все уровни абстракции, необходимые реактивному хранилищу MongoDB.

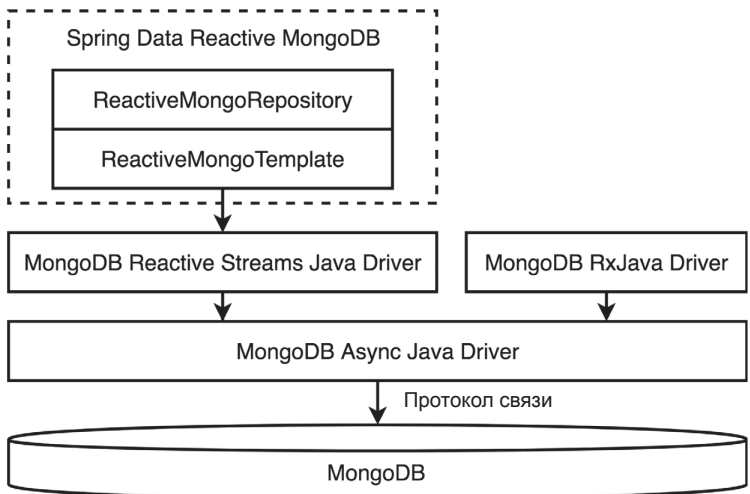


Рис. 7.16. Реактивный стек MongoDB с использованием Spring Data

Использование асинхронных драйверов (Cassandra)

Выше мы рассказали, как организовано реактивное хранилище Mongo на основе реактивного драйвера. Теперь поговорим о том, как реактивное хранилище Cassandra использует асинхронный драйвер.

Подобно `ReactiveMongoRepository`, реактивный коннектор Casandra предлагает нам интерфейс `ReactiveCassandraRepository`, который расширяет более общий интерфейс `ReactiveCrudRepository`. Интерфейс `ReactiveCassandraRepository` реализован классом `SimpleReactiveCassandraRepository`, который, в свою очередь, для выполнения низкоуровневых операций использует интерфейс `ReactiveCassandraOperations`. Интерфейс `ReactiveCassandraOperations` реализован классом `ReactiveCassandraTemplate`. Конечно, `ReactiveCassandraTemplate` можно использовать в приложениях непосредственно как `ReactiveMongoTemplate`.

Класс `ReactiveCassandraTemplate` внутренне использует `ReactiveCqlOperations`. `ReactiveCassandraTemplate` оперирует сущностями Spring Data, такими как `org.springframework.data.cassandra.core.query.Query`, а `ReactiveCqlOperations` – инструкциями на языке CQL (в виде строк `String`), которые распознаются драйвером Cassandra. Интерфейс `ReactiveCqlOperations` реализован классом `ReactiveCqlTemplate`. В свою очередь, для фактического выполнения запросов к базе данных `ReactiveCqlTemplate` использует интерфейс `ReactiveSession`. Интерфейс `ReactiveSession` реализован классом `DefaultBridgedReactiveSession`, который связывает асинхронные методы `Session`, предлагаемые драйвером, с шаблонами реактивного выполнения.

Давайте посмотрим, как класс `DefaultBridgedReactiveSession` превращает асинхронный API в реактивный API. Метод `execute` получает `Statement` (например, с инструкцией `SELECT`) и реактивно возвращает результаты. Метод `execute` и его вспомогательный метод `adaptFuture` выглядят так:

```
public Mono<ReactiveResultSet> execute(Statement statement) { // (1)
    return Mono.create(sink -> { // (2)
        try {
            ListenableFuture<ResultSet> future = this.session // (3)
                .executeAsync(statement);
            ListenableFuture<ReactiveResultSet> resultSetFuture =
                Futures.transform( // (4)
                    future, DefaultReactiveResultSet::new);
            adaptFuture(resultSetFuture, sink); // (5)
        } catch (Exception cause) {
            sink.error(cause); // (6)
        }
    });
}
```

```
<T> void adaptFuture(                                     // (7)
    ListenableFuture<T> future, MonoSink<T> sink
) {
    future.addListener(() -> {                             // (7.1)
        if (future.isDone()) {
            try {
                sink.success(future.get());                 // (7.2)
            } catch (Exception cause) {
                sink.error(cause);                          // (7.3)
            }
        }
    }, Runnable::run);
}
```

Пояснения к коду.

1. Прежде всего метод `execute` возвращает не `Flux` с результатами, а `Mono` с экземпляром `ReactiveResultSet`. `ReactiveResultSet` обертывает асинхронный `com.datastax.driver.core.ResultSet`, поддерживающий разбиение на страницы, поэтому первая страница с результатами извлекается, когда возвращается экземпляр `ResultSet`, а следующие – только после потребления результатов с первой страницы. `ReactiveResultSet` адаптирует это поведение методом со следующей сигнатурой: `Flux<Row> rows()`.
2. Вызовом метода `create` создается новый экземпляр `Mono`, который откладывает операции до появления подписчика.
3. Асинхронное выполнение запроса в асинхронном экземпляре `Session` драйвера. Обратите внимание, что для возврата результатов драйвер `Cassandra` использует `ListenableFuture` из `Guava`.
4. Асинхронный `ResultSet` заворачивается в реактивный аналог `ReactiveResultSet`.
5. Здесь вызывается вспомогательный метод `adaptFuture`, который отображает `ListenableFuture` в `Mono`.
6. Если возникнут какие-то ошибки, реактивный подписчик проинформирует нас об этом.
7. Метод `adaptFuture` просто добавляет нового слушателя в объект `Future` (7.1), поэтому, когда появляется результат, он генерирует реактивный сигнал `onNext` (7.2). Он же уведомляет подписчика об ошибках выполнения, если таковые появятся (7.3).

Важно отметить, что многостраничный `ResultSet` позволяет вызвать метод `fetchMoreResults` для получения следующей страницы в асинхронном режиме. `ReactiveResultSet` делает это за кулисами, внутри метода `Flux<Row> rows()`. Однако такое решение считается промежуточным, пока `Cassandra` не получит настоящий реактивный драйвер.

На рис. 7.17 показана внутренняя архитектура реактивного модуля Spring Data Cassandra:

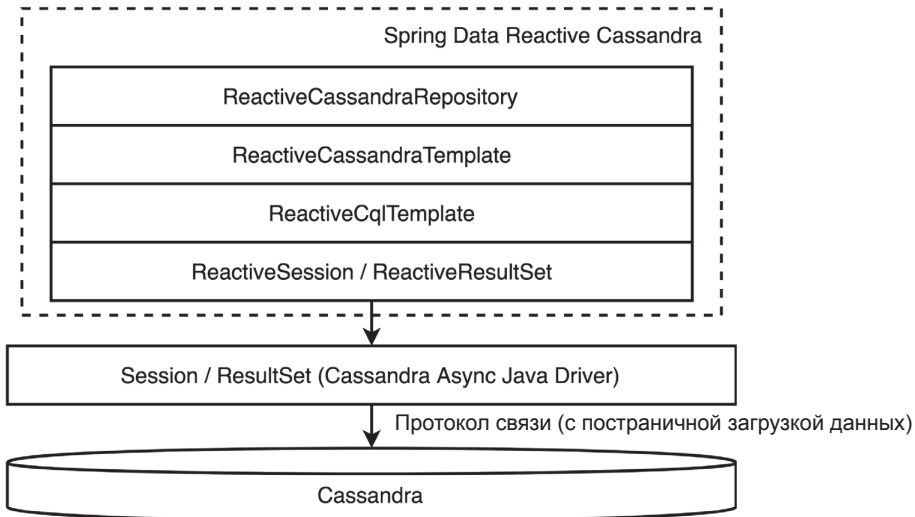


Рис. 7.17. Реактивный стек Cassandra с использованием Spring Data

Реактивные транзакции

Транзакция служит маркером для базы данных, обозначающим границы блока, включающего множество логических операций, которые должны быть выполнены атомарно. Итак, в некоторый момент времени инициализируется новая транзакция, затем с объектом транзакции выполняются некие операции, а потом наступает момент принятия решения. В этот момент клиент и база данных решают зафиксировать или откатить транзакцию.

В синхронном мире объект транзакции часто хранится в контейнере `ThreadLocal`. Однако `ThreadLocal` плохо подходит для использования в реактивных конвейерах, потому что пользователь не может контролировать переключения потоков выполнения. Транзакция требует привязки базового ресурса к материализованному процессу. В Project Reactor для этого лучше всего подходит `Reactor Context`, описанный в главе 4 «*Project Reactor – основа реактивных приложений*».

Реактивные транзакции в MongoDB 4

Начиная с версии 4.0 MongoDB поддерживает *транзакции с несколькими документами*. Это дает нам возможность поэкспериментировать с реактивными транзакциями в новой версии MongoDB. Ранее в Spring Data имелись реактивные коннекторы только для баз данных, не поддерживавших транзакции. Теперь ситуация изменилась. Так как реактивные транзакции являются новинкой в области

реактивных хранилищ, которые сами все еще являются новшеством, все последующие утверждения и примеры кода следует воспринимать как возможные варианты реализации реактивных транзакций в будущем. На момент написания этих строк в Spring Data отсутствовали любые механизмы, применяющие реактивные транзакции на уровне службы или хранилища. Однако мы можем пользоваться транзакциями на уровне `ReactiveMongoOperations`, реализованном в `ReactiveMongoTemplate`.

Сразу отметим, что транзакции с несколькими документами в MongoDB – это новая функция. Она работает только с не сегментированными наборами реплицируемых данных и механизмом хранения `WiredTiger`. Никакие другие конфигурации не поддерживают транзакции с несколькими документами в MongoDB 4.0.

Также некоторые возможности MongoDB недоступны внутри транзакции. Не допускается выполнять метакоманды или создавать коллекции или индексы. Также внутри транзакций не работает неявное создание коллекций. Поэтому для предотвращения ошибок необходимо настроить структуры базы данных. Кроме того, некоторые команды могут проявлять иное поведение, поэтому обязательно загляните в документацию с описанием многодокументных транзакций.

Прежде MongoDB поддерживала возможность атомарного изменения только для одного документа, хотя такой документ мог содержать вложенные документы. С появлением поддержки многодокументных транзакций стала возможной семантика «все или ничего» для групп операций, документов и коллекций. Многодокументные транзакции гарантируют глобальную непротиворечивость данных. После подтверждения транзакции сохраняются все изменения, выполненные в рамках транзакции. Однако если одна из операций внутри транзакции терпит неудачу, транзакция прерывается, и отменяются все изменения, выполненные к этому моменту. Кроме того, измененные данные не видны за границами транзакции, пока она не будет подтверждена.

Теперь посмотрим, как можно использовать реактивные транзакции для сохранения документов в MongoDB. С этой целью используем известный классический пример. Допустим, нам нужно реализовать службу электронного кошелька, способную переводить деньги между счетами пользователей. У каждого пользователя есть счет с неотрицательным балансом. Пользователь может перевести произвольную сумму другому пользователю, только если имеет достаточно средств. Переводы могут происходить параллельно, но для системы непозволительно терять или получать из ниоткуда деньги при выполнении переводов. Следовательно, операция списания денег со счета отправителя и зачисления денег на счет получателя должны происходить вместе и атомарно. Многодокументные транзакции должны помочь в этом.

Без транзакций мы можем столкнуться со следующими проблемами:

- клиент выполняет сразу несколько переводов на общую сумму, превышающую сумму на его счету. То есть имеется вероятность, что одновременное

выполнение переводов отрицательно повлияет на непротиворечивость состояния системы и породит лишние деньги;

- пользователь получает сразу несколько переводов. Некоторые операции пополнения могут затереть друг друга, и тогда пользователь может потерять деньги навсегда.

Есть несколько алгоритмов перевода денег, но здесь мы будем использовать самый простой из них. Перевод суммы со счета А на счет В будет осуществляться так.

1. Запустить новую транзакцию.
2. Загрузить кошелек для счета А.
3. Загрузить кошелек для счета В.
4. Проверить, достаточно ли средств на счете А.
5. Вычислить новый баланс для счета А после списания денег.
6. Вычислить новый баланс для счета В после зачисления денег.
7. Сохранить изменения в счете А.
8. Сохранить изменения в счете В.
9. Подтвердить транзакцию.

Используя этот алгоритм, получим новое, согласованное состояние кошельков или все останется по-старому.

Давайте опишем класс `Wallet` кошелька, отображаемого в документ MongoDB и имеющего несколько вспомогательных методов.

```
@Document(collection = "wallet") // (1)
public class Wallet {
    @Id private ObjectId id; // (2)
    private String owner;
    private int balance;

    // Конструкторы и методы свойств опущены...

    public boolean hasEnoughFunds(int amount) { // (3)
        return balance >= amount;
    }

    public void withdraw(int amount) { // (4)
        if (!hasEnoughFunds(amount)) {
            throw new IllegalStateException("Not enough funds!");
        }
        this.balance = this.balance - amount;
    }

    public void deposit(int amount) { // (5)
```

```

        this.balance = this.balance + amount;
    }
}

```

Пояснения к коду.

1. Класс сущности `Wallet` отображается в коллекцию `wallet` в MongoDB.
2. Класс `org.bson.types.ObjectId` используется в роли идентификатора сущности. Класс `ObjectId` превосходно интегрируется с MongoDB и часто используется для идентификации сущностей.
3. Метод `hasEnoughFunds` проверяет, достаточно ли средств в кошельке для выполнения операции.
4. Метод `withdraw` уменьшает баланс кошелька на указанную сумму.
5. Метод `deposit` увеличивает баланс кошелька на указанную сумму.

Для сохранения и загрузки кошельков нам необходимо хранилище:

```

public interface WalletRepository
    extends ReactiveMongoRepository<Wallet, ObjectId> {           // (1)
    Mono<Wallet> findByOwner(Mono<String> owner);                 // (2)
}

```

Опишем интерфейс `WalletRepository` подробнее.

1. Наш интерфейс `WalletRepository` наследует интерфейс `ReactiveMongoRepository`.
2. Также определяем дополнительный метод `findByOwner` для получения кошелька по имени владельца. Сгенерированная реализация интерфейса знает, как выполнить фактический запрос, потому что имя метода `findByOwner` следует соглашениям, принятым в Spring Data.

Теперь определим интерфейс `WalletService`.

```

public interface WalletService {

    Mono<TxResult> transferMoney(                                // (1)
        Mono<String> fromOwner,
        Mono<String> toOwner,
        Mono<Integer> amount);

    Mono<Statistics> reportAllWallets();                         // (2)
    enum TxResult {                                             // (3)
        SUCCESS,
        NOT_ENOUGH_FUNDS,
        TX_CONFLICT
    }

    class Statistics {                                           // (4)

```



```

    // Реализация опущена ...
  }
}

```

Пояснения к коду.

1. Метод `transferMoney` переводит указанную сумму из кошелька `fromOwner` в кошелек `toOwner`. Обратите внимание, что метод использует реактивные типы, то есть на момент вызова метода фактические участники транзакции могут оставаться неизвестными. Конечно, метод мог бы точно так же принимать простые типы или `Mono<MoneyTransferRequest>`. Но мы преднамеренно использовали три разных экземпляра `Mono`, чтобы показать работу с оператором `zip` и `TupleUtils`.
2. Метод `reportAllWallets` получает данные из всех зарегистрированных кошельков и проверяет общий баланс.
3. Метод `transferMoney` возвращает результат типа `TxResult`. Перечисление `TxResult` описывает три возможных исхода операции перевода: `SUCCESS` (успех), `NOT_ENOUGH_FUNDS` (недостаточно средств) и `TX_CONFLICT` (конфликт, возникший при переводе). Результаты `SUCCESS` и `NOT_ENOUGH_FUNDS` не требуют дополнительных пояснений. А `TX_CONFLICT` описывает ситуацию, когда транзакция потерпела неудачу, потому что преуспели какие-то другие транзакции, обновившие один или оба кошелька, вовлеченных в эту транзакцию.
4. Класс `Statistics` представляет агрегированное состояние всех кошельков в системе, что очень удобно для выполнения проверки целостности. Детали его реализации опущены ради простоты.

Теперь, определив интерфейс `WalletService`, можем написать модульный тест. Тест принимает желаемое количество параллельных транзакций, для каждой случайно выбирает два кошелька и пытается перевести случайную сумму. Вот как выглядит реализация теста без некоторых малозначительных частей.

```

public Mono<OperationStats> runSimulation() {
    return Flux.range(0, iterations)                                // (1)
        .flatMap(i -> Mono
            .delay(Duration.ofMillis(rnd.nextInt(10)))              // (2)
            .publishOn(simulationScheduler)                         // (3)
            .flatMap(_i -> {
                String fromOwner = randomOwner();                  // (4)
                String toOwner = randomOwnerExcept(fromOwner);    //
                int amount = randomTransferAmount();              //
                return walletService.transferMoney(               // (5)
                    Mono.just(fromOwner),
                    Mono.just(toOwner),
                    Mono.just(amount));
            })
        })
}

```

```

        .reduce(
            OperationStats.start(),
            OperationStats::countTxResult);
    }

```

// (6)

Предыдущий код выполняет следующие шаги.

1. Вызывается метод `Flux.range` для имитации желаемого числа итераций перевода средств.
2. Применяется небольшая случайная задержка, чтобы симитировать случайное конкурентное выполнение транзакций.
3. Транзакции запускаются в `simulationScheduler`. Уровень параллелизма определяется количеством выполняющихся транзакций. В этом коде можно создать план вызовом `Schedulers.newParallel("name", parallelism)`.
4. Случайный выбор кошельков и сумм для перевода.
5. Выполнение запроса к службе `transferMoney`.
6. Поскольку вызов `transferMoney` может привести к одному из состояний `TxResult`, метод `reduce` помогает отследить статистику моделирования. Обратите внимание, что класс `OperationStats` подчитывает количество успешных операций, отвергнутых из-за недостатка средств на счете и неудавшихся из-за конфликтов между транзакциями. Класс `WalletService.Statistics`, в свою очередь, следит за общим балансом.

При правильной реализации `WalletService` ожидается, что в результате моделирования мы получим состояние системы, в котором общая сумма средств на счетах не изменится. В то же время ожидается, что переводы денег будут выполнены успешно, если у отправителя достаточно средств на счету. Иначе мы столкнемся с нарушениями целостности системы, которые могут привести к фактическим финансовым потерям.

Теперь реализуем службу `WalletService`, используя реактивную поддержку транзакций, предлагаемую `MongoDB 4` и `Spring Data`. Реализация в виде класса `TransactionalWalletService` представлена ниже.

```

public class TransactionalWalletService implements WalletService {
    private final ReactiveMongoTemplate mongoTemplate;

    @Override
    public Mono<TxResult> transferMoney(
        Mono<String> fromOwner,
        Mono<String> toOwner,
        Mono<Integer> requestAmount
    ) {
        return Mono.zip(fromOwner, toOwner, requestAmount)
            .flatMap(function((from, to, amount) -> {
                return doTransferMoney(from, to, amount)
            })

```

// (1)

// (2)

//

//

//

//

// (2.1)

// (2.2)

// (2.3)

```

        .retryBackoff(
            20, Duration.ofMillis(1),
            Duration.ofMillis(500), 0.1
        )
        .onErrorReturn(TxResult.c);
    }));
}

private Mono<TxResult> doTransferMoney(
    String from, String to, Integer amount
) {
    return mongoTemplate.inTransaction().execute(session ->
        session
        .findOne(queryForOwner(from), Wallet.class)
        .flatMap(fromWallet -> session
            .findOne(queryForOwner(to), Wallet.class)
            .flatMap(toWallet -> {
                if (fromWallet.hasEnoughFunds(amount)) {
                    fromWallet.withdraw(amount);
                    toWallet.deposit(amount);
                    return session.save(fromWallet)
                        .then(session.save(toWallet))
                        .then(Mono.just(TxResult.SUCCESS));
                } else {
                    return Mono.just(TxResult.NOT_ENOUGH_FUNDS);
                }
            })
        .onErrorResume(e ->
            Mono.error(new RuntimeException("Conflict")))
        .last();
    }

private Query queryForOwner(String owner) {
    return Query.query(new Criteria("owner").is(owner));
}
}

```

Это не самый простой код, поэтому подробно разберем его.

1. Прежде всего мы обязаны использовать класс `ReactiveMongoTemplate`, потому что на момент написания этих строк реактивный коннектор MongoDB не поддерживал транзакции на уровне хранилищ, а поддерживал только на уровне шаблона MongoDB.
2. Здесь определяется реализация метода `transferMoney`. С помощью операции `zip` он подписывается на все аргументы (2.1), а после получения данных из всех аргументов использует статическую вспомогательную функцию `TupleUtils.function` для разложения `Tuple3<String, String, Integer>`

на составляющие (2.2) для последующего использования. В строке (2.3) вызывается метод `doTransferMoney`, который выполняет фактический перевод денег. Однако `doTransferMoney` может вернуть сигнал `onError`, указывающий на конфликт с другой транзакцией. В этом случае мы повторно пытаемся выполнить транзакцию вызовом метода `retryBackoff` (2.4). Методу `retryBackoff` нужно передать число попыток (20), начальную задержку (1 мс), максимальную задержку между попытками (500 мс) и величину флуктуаций (0.1), определяющую скорость увеличения задержки перед каждой попыткой. Если после всех повторных попыток выполнить транзакцию так и не удалось, мы должны вернуть клиенту код состояния `TX_CONFLICT`.

3. Метод `doTransferMoney` пытается выполнить фактический перевод денег. Он вызывается с уже готовыми аргументами `form`, `to` и `amount` (3.1). Вызовом метода `mongoTemplate.inTransaction().execute(...)` определяем границы новой транзакции (3.2). Внутри метода `execute` получаем `session` – экземпляр класса `ReactiveMongoOperations`. Объект `session` связан с транзакцией `MongoDB`. Теперь, находясь в рамках транзакции, отыскиваем сначала кошелек отправителя (3.3), а затем кошелек получателя (3.4). После получения обоих кошельков проверяем, достаточно ли средств на счете отправителя (3.5). Списываем требуемую сумму со счета отправителя (3.6) и вносим ее на счет получателя (3.7). В этот момент изменения еще не сохранены в базе данных, поэтому сохраняем измененный кошелек отправителя (3.8) и затем получателя (3.9). Если база данных не отвергла изменения, возвращаем код состояния `SUCCESS` и автоматически подтверждаем транзакцию (3.10). Если на счету отправителя недостаточно средств, возвращаем код состояния `NOT_ENOUGH_FUNDS` (3.11). Если в процессе взаимодействий с базой данных возникли какие-то ошибки, мы посылаем сигнал `onError` (3.12), который, в свою очередь, запустит логику повторных попыток, о которой рассказывалось в описании строки (2.4).
4. В строках (3.3) и (3.4) вызываем метод `queryForOwner`, который использует `Criteria API` для конструирования запросов к базе данных `MongoDB`.

Получение ссылок на сеанс с транзакцией реализовано с помощью `Reactor Context`. Метод `ReactiveMongoTemplate.inTransaction` запускает новую транзакцию и помещает ее в контекст. Благодаря этому сеанс с транзакцией, представленный интерфейсом `com.mongodb.reactivestreams.client.ClientSession`, можно получить из любой точки в реактивном конвейере. Вспомогательный метод `ReactiveMongoContext.getSession()` позволяет получить экземпляр сеанса.

Конечно, мы можем улучшить класс `TransactionalWalletService`, организовав получение обоих кошельков в одном запросе и их изменение в одном запросе. Это уменьшит количество запросов к базе данных, повысит скорость перевода денег и уменьшит вероятность конфликтов транзакций. Однако мы оставляем реализацию этих улучшений вам как самостоятельное упражнение.

Теперь можно попробовать запустить прежде описанный сценарий тестирования с другим числом кошельков и количеством переводов. Если вся бизнес-логика в классе `TransactionalWalletService` реализована правильно, мы должны получить следующий вывод:

```
The number of accounts in the system: 500
The total balance of the system before the simulation: 500,000$
Running the money transferring simulation (10,000 iterations)
...
The simulation is finished
Transfer operations statistic:
  - successful transfer operations: 6,238
  - not enough funds: 3,762
  - conflicts: 0
All wallet operations:
  - total withdraw operations: 6,238
  - total deposit operations: 6,238
The total balance of the system after the simulation: 500,000$
```

Итак, в этом примере мы симитировали 10 000 операций перевода, 6238 из которых завершились успехом, а 3762 потерпели неудачу из-за недостаточности средств. Также наша стратегия с повторными попытками позволила разрешить все конфликты между транзакциями, то есть не обнаружилось ни одной транзакции, завершившейся с кодом состояния `TX_CONFLICT`. Как можно судить по результатам в журнале, система сохранила неизменным общий баланс – общую сумму денег в системе. Следовательно, мы обеспечили целостность системы, выполняя переводы денег в конкурентной манере и используя реактивные транзакции MongoDB.

Поддержка многодокументных транзакций для реплицируемых наборов теперь позволяет писать совершенно новые приложения с MongoDB как основной базой данных. Конечно, в будущих версиях MongoDB могут быть реализованы транзакции в сегментированных окружениях и с разными уровнями изоляции. Однако мы должны заметить, что многодокументные транзакции влекут за собой более высокие затраты и увеличение задержки ответа в сравнении с простыми операциями записи документов.

Несмотря на то что реактивные транзакции пока не получили широкого распространения, примеры ясно показывают, что транзакции можно с успехом использовать в реактивном программировании. Реактивные транзакции будут пользоваться высоким спросом при применении реактивных подходов к реляционным базам данных, таким как PostgreSQL. Однако для этого необходим реактивный API доступа к базам данных, которого на момент написания этих строк еще не было.

Распределенные транзакции с шаблоном SAGA

Как отмечалось выше в этой главе, распределенные транзакции можно реализовать разными способами. Конечно, это утверждение верно также для реализации слоя хранения с использованием реактивной парадигмы. Однако, учитывая, что Spring Data поддерживает реактивные транзакции только для MongoDB 4 и вышеупомянутая поддержка транзакций не совместима с **Java Transaction API (JTA)**, отметим, что единственный жизнеспособный вариант реализации распределенных транзакций в реактивных микросервисах – это шаблон SAGA, описанный ранее. Кроме всего прочего, шаблон SAGA имеет хороший потенциал для масштабирования и лучше подходит для реактивных систем, чем альтернативные шаблоны, требующие распределенных транзакций.

Реактивные коннекторы в Spring Data

На момент написания этих строк в Spring Data 2.1 имелись коннекторы для четырех баз данных NoSQL: MongoDB, Cassandra, Couchbase и Redis. Весьма вероятно, что в Spring Data также появится поддержка других хранилищ данных, особенно поддерживающих взаимодействия по протоколу HTTP с использованием Spring WebFlux WebClient.

Мы не будем рассматривать здесь все особенности реактивных коннекторов из Spring Data или детали их реализации. В предыдущих разделах мы рассмотрели многое из этого для MongoDB и Cassandra. Тем не менее остановимся на основных отличительных особенностях каждого реактивного коннектора.

Реактивный коннектор MongoDB

Как описывалось выше в этой главе, в Spring Data имеется превосходная поддержка MongoDB. Модуль Spring Data Reactive MongoDB можно подключить с помощью модуля начальной настройки `spring-boot-starter-data-mongodb-reactive` из Spring Boot. Реактивная поддержка MongoDB предлагает реактивное хранилище. Интерфейс `ReactiveMongoRepository` определяет базовый контракт хранения. Хранилище наследует все особенности `ReactiveCrudRepository` и добавляет поддержку QBE. Также хранилище MongoDB поддерживает пользовательские запросы, определяемые в аннотации `@Query`, и дополнительные настройки запросов в аннотации `@Meta`. Хранилище MongoDB поддерживает динамическое создание запросов из имен методов, если те следуют принятым соглашениям.

Еще одна особенность хранилища MongoDB – поддержка хвостовых курсоров (`tailable cursors`). По умолчанию база данных автоматически закрывает курсор запроса после получения клиентом всех результатов. Однако в MongoDB имеются ограниченные (`capped`) коллекции, которые имеют фиксированный размер и поддерживают операции с высокой пропускной способностью. Извлечение докумен-

та базируется на порядке вставки. Ограниченные коллекции работают подобно циклическим буферам, а кроме того, поддерживают хвостовые курсоры. Этот курсор остается открытым, после того как клиент извлечет все данные, соответствующие начальному запросу, а когда кто-то добавит новые документы в ограниченную коллекцию, хвостовой курсор вернет их. В `ReactiveMongoRepository` метод, отмеченный аннотацией `@Tailable`, возвращает хвостовой курсор, представленный типом `Flux<Entity>`.

Уровнем ниже находится интерфейс `ReactiveMongoOperations` и класс `ReactiveMongoTemplate` с его реализацией, дающий более детальный доступ к взаимодействиям с MongoDB. Помимо прочего, `ReactiveMongoTemplate` поддерживает многодокументные транзакции MongoDB. Эта поддержка работает только для реплицируемых наборов без сегментирования и механизма хранения `WiredTiger`. Она подробно была описана выше, в разделе «*Реактивные транзакции в MongoDB 4*».

Модуль `Spring Data Reactive MongoDB` построен на основе реактивного драйвера `Reactive Streams MongoDB Driver`, который реализует стандарт `Reactive Streams` и внутренне использует библиотеку `Project Reactor`. В свою очередь, реактивный драйвер MongoDB для Java основывается на асинхронном драйвере MongoDB. В разделе «*Как работают реактивные хранилища*» мы описали работу `ReactiveMongoRepository` во всех подробностях.

Реактивный коннектор Cassandra

Модуль `Spring Data Reactive Cassandra` можно подключить, импортировав модуль начальной настройки `spring-boot-starter-data-cassandra-reactive`. Cassandra также имеет поддержку реактивных хранилищ. Интерфейс `ReactiveCassandraRepository` наследует `ReactiveCrudRepository` и определяет слой доступа к базе данных Cassandra. Аннотация `@Query` позволяет вручную определять запросы на языке CQL3. С помощью аннотации `@Consistency` можно настроить желаемый уровень согласованности, применяемый к запросу.

Интерфейс `ReactiveCassandraOperations` и класс `ReactiveCassandraTemplate` открывают доступ к низкоуровневым операциям в базе данных Cassandra.

Начиная с версии `Spring Data 2.1` реактивный коннектор Cassandra основывается на асинхронном драйвере Cassandra. В разделе «*Использование асинхронных драйверов (Cassandra)*» мы описали, как асинхронные взаимодействия вписываются в реализацию реактивного клиента.

Реактивный коннектор Couchbase

Модуль `Spring Data Reactive Couchbase` можно подключить, импортировав модуль начальной настройки `spring-boot-starter-data-couchbase-reactive`. Он обеспечивает реактивный доступ к базе данных Couchbase (<https://www.couchbase.>

com). Интерфейс `ReactiveCouchbaseRepository` расширяет базовый интерфейс `ReactiveCrudRepository`, дополнительно требуя указать тип идентификатора сущности, чтобы расширить интерфейс `Serializable`.

Реализация по умолчанию интерфейса `ReactiveCouchbaseRepository` основана на интерфейсе `RxJavaCouchbaseOperations`, который реализует класс `RxJavaCouchbaseTemplate`. На данный момент для вас должно быть очевидно, что реактивный коннектор Couchbase использует библиотеку RxJava для реализации `RxJavaCouchbaseOperations`. Так как методы `ReactiveCouchbaseRepository` возвращают типы `Mono` и `Flux`, а методы `RxJavaCouchbaseOperations` возвращают тип `Observable`, необходимо выполнять преобразование реактивных типов. Такие преобразования выполняются на уровне реализации хранилища.

Реактивный коннектор Couchbase основан на реактивном драйвере Couchbase. Последний драйвер Couchbase 2.6.2 использует версию RxJava 1.3.8, последнюю в ветке 1.x. Поэтому поддержка обратного давления может ограничиваться коннектором Couchbase. Однако он имеет полностью неблокируемый стек благодаря использованию фреймворка Netty и библиотеки RxJava, соответственно, он не должен понапрасну расходовать ресурсы приложения.

Реактивный коннектор Redis

Подключить модуль Spring Data Reactive Redis можно, импортировав модуль начальной настройки `spring-boot-starter-data-redis-reactive`. В отличие от других реактивных коннекторов, коннектор Redis не предлагает реактивного хранилища. Поэтому главной абстракцией для реактивного доступа к Redis является класс `ReactiveRedisTemplate`. Этот класс реализует API, определяемый интерфейсом `ReactiveRedisOperations`, и предлагает все необходимые процедуры сериализации/десериализации. В то же время `ReactiveRedisConnection` позволяет работать с исходными буферами байтов при взаимодействии с Redis.

Наряду с обычными операциями сохранения и извлечения объектов и управления структурами данных Redis шаблон позволяет подписываться на каналы издатель/подписчик. Например, метод `convertAndSend(String destination, V message)` публикует указанное сообщение в указанном канале и возвращает число клиентов, получивших его. Метод `listenToChannel(String... channels)` возвращает `Flux` с сообщениями из заданных каналов. То есть реактивный коннектор Redis не только поддерживает реактивное хранилище данных, но и предлагает механизм обмена сообщениями. В главе 8 «Масштабирование с Cloud Streams» мы расскажем больше о том, как поддержка сообщений улучшает масштабирование и эластичность реактивного приложения.

В настоящее время Spring Data Redis интегрируется с драйвером Lettuce (<https://github.com/lettuce-io/lettuce-core>). Это единственный реактивный коннектор для Redis. Для реализации внутренних механизмов Lettuce 4.x используется

библиотека RxJava. Однако ветка 5.x была переведена на использование библиотеки Project Reactor.

Все реактивные коннекторы, кроме Couchbase, имеют средства проверки состояния. Поэтому проверки состояния базы данных тоже не должны напрасно расходовать серверные ресурсы. Подробнее об этом рассказывается в главе 10 «*И наконец, выпуск!*».

Мы верим, что со временем в Spring Data появятся и другие реактивные коннекторы.

Ограничения и ожидаемые улучшения

Область реактивных взаимодействий относительно нова, поэтому есть некоторые ограничения, препятствующие использованию реактивного подхода во многих приложениях:

- **отсутствие реактивных драйверов** для популярных баз данных, широко используемых в современных проектах. В настоящее время у нас есть реактивные или асинхронные драйверы только для MongoDB, Cassandra, Redis и Couchbase. Соответственно, для этих баз данных имеются реактивные коннекторы в экосистеме Spring Data. Также у нас есть некоторые варианты реактивного доступа к PostgreSQL. В то же время продолжается работа над поддержкой реактивного доступа для MySQL и MariaDB. Несмотря на то что наличие нескольких баз данных с поддержкой реактивности может удовлетворить немалые потребности, этот список все еще слишком ограничен. Чтобы заслужить популярность в среде разработчиков, реактивный доступ к данным должен быть поддержан появлением коннекторов для более широкого круга баз данных, таких как PostgreSQL, MySQL, Oracle, MS SQL, популярных поисковых механизмов, таких как Elasticsearch и Apache Solr, а также облачных баз данных, таких как Google Big Query, Amazon Redshift и Microsoft CosmosDB;
- **отсутствие поддержки реактивности в JPA.** В настоящее время поддержка реактивности работает на очень низком уровне. Очень непросто организовать реактивный доступ к сущностям, используя приемы, предлагаемые обычным JPA. Ныне существующие реактивные коннекторы не поддерживают возможности отображения отношений между сущностями, кеширования сущностей или отложенной загрузки. С другой стороны, было бы странно требовать подобные возможности при отсутствии низкоуровневого API для реактивного доступа к данным;
- **отсутствие реактивного API для доступа к данным на уровне языка.** Как отмечалось выше в этой главе, на момент написания книги платформа Java предлагала для доступа к данным только JDBC API, блокирующий и синхронный по своей природе, который по этой причине невозможно бесшовно использовать в реактивных приложениях.

Однако мы можем видеть, что все большее число решений NoSQL предлагает реактивные драйверы или, по крайней мере, асинхронные, которые легко интегрируются в реактивные API. Кроме того, в настоящее время вносятся серьезные улучшения в API доступа к данным на уровне языка Java. На момент написания этих строк было сделано два важных предложения, восполняющих этот недостаток: ADBA и R2DBC. Поэтому мы предлагаем рассмотреть их более внимательно.

Асинхронный доступ к базам данных

Асинхронный доступ к базе данных (Asynchronous Database Access, ADBA) определяет неблокирующий API доступа к базе данных для платформы Java. На момент написания этих строк данное предложение все еще находилось в состоянии проектирования, и в экспертной комиссии JDBC Expert Group обсуждали, как должен выглядеть этот API. Предложение ADBA было анонсировано на конференции JavaOne в 2016 году и обсуждается уже несколько лет. Цель ADBA – дополнить текущий JDBC API и предложить асинхронную альтернативу (а не замену) для программ с высокой пропускной способностью. ADBA предполагает использовать поддержку текучего (fluent) стиля программирования и шаблон проектирования «Компоновщик» (Builder) для составления запросов к базе данных. ADBA не является расширением JDBC и не зависит от него. Когда все будет готово, ADBA, скорее всего, будет находиться в пакете `java.sql2`.

ADBA – это асинхронный API, поэтому никакие вызовы методов не должны блокироваться при выполнении сетевых запросов. Все потенциально блокирующие действия определены как отдельные операции ADBA. Клиентское приложение создает и отправляет одну операцию или граф операций ADBA. Драйвер, реализующий ADBA, асинхронно выполняет операцию и сообщает результат через `java.util.concurrent.CompletionStage` или обратные вызовы. Когда это предложение будет реализовано, асинхронный запрос через ADBA будет выглядеть примерно так:

```
CompletionStage<List<String>> employeeNames =           // (1)
    connection
        .<Integer>rowOperation("select * from employee") // (2)
        .onError(this::userDefinedErrorHandler)         // (3)
        .collect(Collector.of(                          // (4)
            () -> new ArrayList<String>(),
            (ArrayList<String> cont, Result.RowColumn row) ->
                cont = cont.add(row.at("name").get(String.class)),
            (l, r) -> l,
            container -> container))
        .submit()                                       // (5)
        .getCompletionStage();                         // (6)
```

Имейте в виду, что предыдущий код основан на проектных предложениях ADBA, поэтому фактический API может измениться. Вот некоторые замечания к этому коду.

1. Запрос возвращает результат типа `CompletionStage`, в данном случае список имен сотрудников.
2. Вызовом метода `rowOperation` соединения `connection` с базой данных создается новая операция.
3. Для операции можно зарегистрировать обработчик ошибки вызовом метода `onError`. Обработка ошибки осуществляется нашим методом `userDefinedErrorHandler`.
4. Метод `collect` собирает результаты с помощью `Java Stream API`.
5. Метод `submit` запускает операцию.
6. Метод `getCompletionStage` возвращает пользователю экземпляр `CompletionStage`, который будет хранить результаты по завершении обработки запроса.

Конечно, ADBA даст возможность писать и выполнять более сложные запросы к базе данных, включая условные, параллельные и взаимозависимые операции. ADBA поддерживает транзакции. Однако, в отличие от JDBC, ADBA не предназначен для прямого использования бизнес-логикой (даже если это возможно), его цель – предоставить асинхронную основу для более высокоуровневых библиотек и фреймворков.

На данный момент имеется только одна реализация ADBA, которая называется **AoJ**. AoJ (<https://gitlab.com/asyncjdbc/asyncjdbc/tree/master>) – это экспериментальная библиотека, реализующая подмножество ADBA API посредством вызовов стандартного JDBC в отдельном пуле потоков. Библиотека AoJ пока не готова к промышленному использованию, но дает возможность поиграть с ADBA без необходимости реализовать полноценный асинхронный драйвер.

Существуют некоторые предложения возвращать из ADBA результаты не только в виде `CompletionStage`, но и в виде реактивного издателя `Publisher` из `Java Flow API`. Однако пока не ясно, будет ли `Flow API` интегрирован в ADBA, или ADBA предложит свой реактивный API. Эта тема – все еще предмет горячих споров.

В этой точке мы должны еще раз заявить, что асинхронное поведение, представляемое `Java`-типом `CompletionStage`, всегда можно заменить реактивной реализацией `Publisher`. Однако обратное утверждение неверно. Реактивное поведение можно реализовать на основе `CompletionStage` или `CompletableFuture` только с некоторыми допущениями, а именно отказом от поддержки обратного давления. Кроме того, в `CompletionStage <List <T>>` отсутствует фактическая семантика потоковой передачи данных, и клиент вынужден ждать поступления полного набора результатов. Использование `Collector API` для потоковой передачи, похоже, здесь не подходит.

Кроме того, `CompletableFuture` начинает выполнение сразу после вызова `submit`, тогда как `Publisher` – только после появления подписчика. Реактивный API для доступа к базе данных удовлетворяет требованиям обоих API уровня языка – реактивного и асинхронного, потому что любой реактивный API можно быстро превратить в асинхронный без каких-либо семантических компромиссов. Однако асинхронный API в большинстве случаев можно превратить в реактивный, только с некоторыми компромиссами. Вот почему ADBA с поддержкой Reactive Streams представляется авторам этой книги более выгодным, чем только асинхронная версия ADBA.

Альтернативный кандидат на API доступа к данным следующего поколения в Java называется R2DBC. Он предлагает больше, чем асинхронный ADBA, и доказывает, что реактивный API для доступа к реляционным данным имеет огромный потенциал. Итак, давайте рассмотрим его более внимательно.

Реактивное соединение с реляционной базой данных

Реактивное соединение с реляционной базой данных (Reactive Relational Database Connectivity, R2DBC) (<https://github.com/r2dbc>) – это инициатива для изучения, как может выглядеть полностью реактивный API доступа к базе данных. Команда Spring Data возглавляет инициативу R2DBC и использует ее для опробования и проверки идей в контексте реактивного доступа к данным в реактивных приложениях. Инициатива R2DBC была анонсирована на конференции Spring OnePlatform 2018 и имеет целью определить реактивный API доступа к базе данных с поддержкой обратного давления. Члены команды Spring Data приобрели уникальный опыт работы с реактивными хранилищами NoSQL, поэтому они решили представить свой взгляд на реактивный API доступа на уровне языка. Кроме того, R2DBC может стать базовым API для реактивных реляционных хранилищ в Spring Data. В настоящее время реализация R2DBC все еще находится на стадии экспериментов, и пока не ясно, когда она станет программным обеспечением, готовым к использованию (если это вообще возможно).

Проект R2DBC состоит из следующих частей:

- **R2DBC Service Provider Interface (SPI)** – определяет минимальный API для реализации драйверов. API очень краткий, чтобы уменьшить объем требований к разработчикам драйверов. SPI не предназначен для непосредственного использования в приложениях. Эту задачу должна выполнять выделенная клиентская библиотека;
- **R2DBC Client** – предлагает удобный API и вспомогательные классы для преобразования пользовательских запросов на уровень SPI. Этот отдельный уровень абстракции добавляет некоторый комфорт при непосредственном использовании R2DBC. Авторы подчеркивают, что клиент R2DBC для R2DBC

SPI – это то же самое, что библиотека Jdbi для JDBC. Однако любой может свободно использовать SPI напрямую или реализовать свою клиентскую библиотеку через R2DBC SPI;

- **R2DBC PostgreSQL Implementation** – определяет драйвер R2DBC для PostgreSQL. Он использует платформу Netty для асинхронной связи через протокол связи с PostgreSQL. Управление обратным давлением может осуществляться либо с помощью TCP Flow Control, либо посредством функции в PostgreSQL, называемой порталом, которая фактически является курсором в запросе. Портал имеет прекрасную совместимость со стандартом Reactive Streams. Важно отметить, что не все реляционные базы данных имеют функции протокола связи, необходимые для правильной передачи обратного давления. Однако по крайней мере TCP Flow Control доступен во всех случаях.

Вот как R2DBC Client позволяет работать с базой данных PostgreSQL:

```
PostgresqlConnectionFactory pgConnectionFactory = // (1)
    PostgresqlConnectionFactory.builder()
        .host("<host>")
        .database("<database>")
        .username("<username>")
        .password("<password>")
        .build();

R2dbc r2dbc = new R2dbc(pgConnectionFactory); // (2)

r2dbc.inTransaction(handle -> // (3)
    handle
        .execute("insert into book (id, title, publishing_year) " + // (3.1)
            "values ($1, $2, $3)",
            20, "The Sands of Mars", 1951)
        .doOnNext(n -> log.info("{} rows inserted", n)) // (3.2)
    ).thenMany(r2dbc.inTransaction(handle -> // (4)
        handle.select("SELECT title FROM book") // (4.1)
        .mapResult(result -> // (4.2)
            result.map((row, rowMetadata) -> // (4.3)
                row.get("title", String.class)))) // (4.4)
        .subscribe(elem -> log.info(" - Title: {}", elem)); // (5)
```

Рассмотрим подробнее, как работает этот код.

1. Прежде всего мы должны настроить фабрику соединений, представленную классом `PostgresqlConnectionFactory`. Настройка осуществляется просто.
2. Мы должны создать экземпляр класса `R2dbc`, предлагающего реактивный API.
3. `R2dbc` позволяет создать транзакцию вызовом метода `inTransaction`. Декоратор `handle` обертывает экземпляр реактивного соединения и предоставляет дополнительный удобный API. Здесь мы можем выполнить инструкцию SQL, например вставить новую запись. Метод `execute` получает запрос

SQL и его параметры, если таковые имеются (3.1), и возвращает количество затронутых записей. В предыдущем коде выводим количество обновленных записей (3.2).

4. Вставив запись, запускаем другую транзакцию, чтобы выбрать названия всех книг (4.1). Когда результат получен, отображаем отдельные записи (4.2) с помощью функции `map` (4.3), которая знает о структуре записи. В нашем случае извлекаем поле `title` типа `String` (4.4). Метод `mapResult` возвращает тип `Flux<String>`.
5. Реактивно регистрируем все сигналы `onNext`. Каждый сигнал сопровождается названием книги, включая добавленное на шаге (3).

Как видите, R2DBC Client поддерживает текущий API в реактивном стиле. Этот API соотрится очень естественно в реактивных приложениях.

Использование R2DBC вместе с Spring Data R2DBC

Конечно, команда Spring Data не смогла устоять перед соблазном реализовать интерфейс `ReactiveCrudRepository` поверх R2DBC. На момент написания этих строк данная реализация находилась в модуле Spring Data JDBC, который описан выше в этой главе. Однако они собираются перенести ее в отдельный модуль под названием **Spring Data R2DBC**. Класс `SimpleR2dbcRepository` реализует интерфейс `ReactiveCrudRepository`, используя механизмы R2DBC. Стоит отметить, что по умолчанию класс `SimpleR2dbcRepository` не использует клиента R2DBC Client и определяет своего клиента для работы с SPI R2DBC.

До появления Spring Data R2DBC реактивная поддержка в модуле Spring Data JDBC будет находиться в ветке `Git r2dbc` проекта и пока еще не готова к промышленному использованию. Однако модуль Spring Data JDBC с поддержкой R2DBC демонстрирует огромный потенциал использования `ReactiveCrudRepository` для манипулирования реляционными данными. Давайте попробуем определить наш первый `ReactiveCrudRepository` для PostgreSQL.

```
public interface BookRepository
    extends ReactiveCrudRepository<Book, Integer> {

    @Query("SELECT * FROM book WHERE publishing_year = " +
           "(SELECT MAX(publishing_year) FROM book)")
    Flux<Book> findTheLatestBooks();
}
```

В настоящее время модуль Spring Data JDBC не поддерживает автоматическую конфигурацию, поэтому мы должны вручную создать экземпляр интерфейса `BookRepository`.

```
BookRepository createRepository(PostgresqlConnectionFactory fct) { // (1)
    TransactionalDatabaseClient txClient =                               // (2)
```

```
TransactionalDatabaseClient.create(fct);  
RelationalMappingContext cnt = new RelationalMappingContext(); // (3)  
return new R2dbcRepositoryFactory(txClient, cnt)                // (4)  
    .getRepository(BookRepository.class);                      // (5)  
}
```

В предыдущем коде выполняются следующие шаги.

1. Нам нужна ссылка на экземпляр `PostgresqlConnectionFactory`, созданный в предыдущем примере.
2. `TransactionalDatabaseClient` предлагает базовую поддержку транзакций.
3. Мы должны создать контекст `RelationalMappingContext`, чтобы отображать записи в сущности и обратно.
4. Создается соответствующая фабрика хранилищ. Класс `R2dbcRepositoryFactory` знает, как создать `ReactiveCrudRepository`.
5. Фабрика генерирует экземпляр интерфейса `BookRepository`.

Теперь используем наш реактивный `BookRepository` в обычном реактивном конвейере.

```
bookRepository.findTheLatestBooks()  
    .doOnNext(book -> log.info("Book: {}", book))  
    .count()  
    .doOnSuccess(count -> log.info("DB contains {} latest books", count))  
    .subscribe();
```

Несмотря на то что проект R2DBC все еще экспериментальный и поддерживается в Spring Data JDBC, можем видеть, что по-настоящему реактивный доступ к данным не за горами. Кроме того, проблема управления обратным давлением решена на уровне R2DBC SPI.

Пока не ясно, получит ADBA реактивную поддержку или R2DBC станет реактивной альтернативой ADBA. Однако в обоих случаях есть уверенность, что реактивный доступ к реляционным данным очень скоро станет реальностью, по крайней мере для баз данных с драйверами, совместимыми с ADBA или R2DBC.

Преобразование синхронного хранилища в реактивное

Хотя Spring Data предоставляет реактивные коннекторы для популярных баз данных NoSQL, реактивному приложению иногда приходится обращаться к базе данных, не имеющей реактивного коннектора. Любые блокирующие взаимодействия

ствия в принципе можно завернуть в реактивный API. Правда, такие взаимодействия должны происходить в соответствующем пуле потоков, иначе мы заблокируем цикл обработки событий приложения и полностью его остановим. Обратите внимание, что небольшой пул потоков (с ограниченной очередью), вероятно, будет исчерпан в какой-то момент. В определенный момент заполненная очередь перейдет в состояние блокировки, и тогда смысл ее использования потеряется. Такие решения менее эффективны, чем полностью реактивные аналоги. Однако подход с выделенным пулом потоков для блокировки запросов иногда вполне можно использовать в реактивном приложении.

Предположим, мы должны реализовать реактивный микросервис, который время от времени посылает запросы реляционной базе данных. Эта база данных имеет драйвер JDBC, но не имеет асинхронного или реактивного драйвера. В этом случае единственным вариантом будет создание реактивного адаптера, который скрывает блокирующие запросы за реактивным API.

Как отмечалось выше, все блокирующие запросы должны выполняться под управлением выделенного планировщика. Базовый пул потоков определяет уровень параллелизма для блокирующих действий. Например, при выполнении блокирующих действий в `Schedulers.elastic()` количество *одновременных* запросов не ограничивается, поскольку планировщик `elastic` не ограничивает максимальное количество созданных пулов потоков. В то же время `Scheduler.newParallel("jdbc", 10)` определяет количество рабочих потоков выполнения в пуле, поэтому одновременно может выполняться не более десяти запросов. Этот подход хорошо работает, когда взаимодействия с базой данных выполняются через пул соединений фиксированного размера. В большинстве случаев не имеет смысла устанавливать размер пула потоков выполнения больше, чем размер пула соединений. Например, в планировщике, работающем с неограниченным пулом потоков, в случае исчерпания пула соединений новая задача и выполняющий ее поток будут заблокированы не сетевым обменом, а на этапе извлечения соединения из пула соединений.

При выборе подходящего блокирующего API есть несколько вариантов. Каждый имеет свои плюсы и минусы. Здесь мы рассмотрим библиотеку `rxjava2-jdbc` и узнаем, как обернуть существующее блокирующее хранилище.

С помощью библиотеки `rxjava2-jdbc`

Библиотека `rxjava2-jdbc` (<https://github.com/davidmoten/rxjava2-jdbc>) была создана Дэвидом Мотеном (David Moten) с целью обернуть драйвер JDBC так, чтобы он не блокировал реактивное приложение. Библиотека основана на RxJava 2 и использует выделенный пул потоков и идею неблокирующего пула соединений. Соответственно, запрос не блокирует поток во время ожидания свободного соединения. Как только соединение становится доступным, запрос начинает вы-

полняться через соединение и блокирует поток выполнения. Приложение может не использовать выделенного планировщика для блокирующих запросов, потому что библиотека сама решает эту задачу. Кроме того, библиотека имеет текущий предметно-ориентированный язык (DSL), который позволяет запускать запросы SQL и получать результаты в виде реактивных потоков. Давайте определим сущность Book и аннотируем ее для использования с rxjava2-jdbc.

```
@Query("select id, title, publishing_year " + // (3)
      "from book order by publishing_year")
public interface Book { // (1)
    @Column String id(); // (2)
    @Column String title();
    @Column Integer publishing_year();
}
```

Пояснения к коду.

1. Определение интерфейса Book. Обратите внимание, что при использовании Spring Data мы обычно определяем сущность как класс.
2. Методы доступа декорируются аннотацией @Column. Она помогает отобразить столбцы записи в поля сущности.
3. С помощью аннотации @Query определяется инструкция SQL для извлечения сущности.

Теперь определим простое хранилище, выполняющее поиск книг, опубликованных в определенный период.

```
public class RxBookRepository {
    private static final String SELECT_BY_YEAR_BETWEEN = // (1)
        "select * from book where " +
        "publishing_year >= :from and publishing_year <= :to";

    private final String url = "jdbc:h2:mem:db";
    private final int poolSize = 25;
    private final Database database = Database.from(url, poolSize); // (2)

    public Flowable<Book> findByYearBetween( // (3)
        Single<Integer> from,
        Single<Integer> to
    ) {
        return Single
            .zip(from, to, Tuple2::new) // (3.1)
            .flatMapPublisher(tuple -> database //
                .select(SELECT_BY_YEAR_BETWEEN) // (3.2)
                .parameter("from", tuple._1()) // (3.3)
                .parameter("to", tuple._2()) //
                .autoMap(Book.class)); // (3.5)
    }
}
```

```
}  
}
```

Рассмотрим подробнее реализацию класса `RxBookRepository`.

1. Так как библиотека не способна генерировать запросы автоматически, мы должны определить SQL-запрос, который отыщет нужные книги. В запросах допускается использовать именованные параметры.
2. Для инициализации базы данных необходимо указать JDBC URL и размер пула. В нашем случае одновременно может выполняться не более 25 запросов.
3. Метод `findByYearBetween` использует реактивные типы из библиотеки `RxJava 2` (`Flowable` и `Single`), а не из `Project Reactor`. Это связано с тем, что библиотека `rxjava2-jdbc` внутренне использует `RxJava 2.x` и экспортирует типы `RxJava` через свой API. Однако типы `RxJava` легко преобразуются в типы `Project Reactor`. В строке (3.1) мы подписываемся на поток, возвращающий аргументы запроса. Затем вызываем метод `select` (3.2) и заполняем параметры запроса (3.3). Метод `autoMap` преобразует запись JDBC в сущность `Book`. Метод `autoMap` возвращает поток `Flowable<Book>`, эквивалентный потоку из `Project Reactor Flux<Book>`.

Библиотека `rxjava2-jdbc` поддерживает большинство драйверов JDBC. Также в библиотеке есть поддержка транзакций. Все операции внутри транзакции должны выполняться в одном соединении. Подтверждение/откат транзакции происходят автоматически.

Библиотека `rxjava2-jdbc` помогает свести к минимуму некоторые потенциальные блокировки потоков и реактивно работать с реляционными базами данных. Однако она до сих пор является относительно новой и может не обрабатывать сложные реактивные рабочие процессы, особенно включающие транзакции. Библиотека `rxjava2-jdbc` также требует определять все запросы SQL.

Обертывание синхронного `CrudRepository`

Иногда в программе уже может иметься экземпляр `CrudRepository` со всем необходимым для доступа к данным, но его нельзя напрямую использовать в реактивном приложении. В этом случае можно написать собственный реактивный адаптер, который будет действовать подобно библиотеке `rxjava2-jdbc`, но на уровне хранилища. Будьте осторожны, применяя этот подход с JPA. Можно быстро столкнуться с проблемами проксирования при использовании отложенной загрузки. Итак, предположим, у нас есть следующая сущность `Book`, определенная в JPA.

```
@Entity  
@Table(name = "book")  
public class Book {
```

```

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String title;
    private Integer publishingYear;
    // Конструкторы, методы свойств...
}

```

Также у нас есть следующее хранилище Spring Data JPA.

```

@Repository
public interface BookJpaRepository
    extends CrudRepository<Book, Integer> { // (1)

    Iterable<Book> findByIdBetween(int lower, int upper); // (2)

    @Query("SELECT b FROM Book b WHERE " + "LENGTH(b.title)
        = (SELECT MIN(LENGTH(b2.title)) FROM Book b2)") // (3)
    Iterable<Book> findShortestTitle();

}

```

BookJpaRepository обладает следующими характеристиками.

1. Расширяет интерфейс CrudRepository и наследует все методы для доступа к данным.
2. BookJpaRepository определяет метод, генерирующий запрос в соответствии с соглашением об именовании.
3. BookJpaRepository определяет метод с запросом SQL.

Интерфейс BookJpaRepository прекрасно работает с блокирующей инфраструктурой JPA. Все методы BookJpaRepository возвращают неактивные типы. Чтобы завернуть интерфейс BookJpaRepository в реактивный API и получить максимум от его возможностей, можно определить абстрактный адаптер и расширить его дополнительными методами отображения findByIdBetween и findShortestTitle. Абстрактный адаптер можно повторно использовать для преобразования любого экземпляра CrudRepository. Вот как мог бы выглядеть такой адаптер:

```

public abstract class
    ReactiveCrudRepositoryAdapter
    <T, ID, I extends CrudRepository<T, ID>> // (1)
    implements ReactiveCrudRepository<T, ID> { //

    protected final I delegate; // (2)
    protected final Scheduler scheduler; //

    // Конструктор...

    @Override
    public <S extends T> Mono<S> save(S entity) { // (3)

```

```

        return Mono
            .fromCallable(() -> delegate.save(entity)) // (3.1)
            .subscribeOn(scheduler); // (3.2)
    }

    @Override
    public Mono<T> findById(Publisher<ID> id) { // (4)
        return Mono.from(id) // (4.1)
            .flatMap(actualId -> // (4.2)
                delegate.findById(actualId) // (4.3)
                .map(Mono::just) // (4.4)
                .orElseGet(Mono::empty)) // (4.5)
            .subscribeOn(scheduler); // (4.6)
    }

    @Override
    public Mono<Void> deleteAll(Publisher<? extends T> entities) { // (5)
        return Flux.from(entities) // (5.1)
            .flatMap(entity -> Mono //
                .fromRunnable(() -> delegate.delete(entity)) // (5.2)
                .subscribeOn(scheduler)) // (5.3)
            .then(); // (5.4)
    }
    // Все остальные методы ReactiveCrudRepository...
}

```

Пояснения к коду.

1. `ReactiveCrudRepositoryAdapter` – абстрактный класс, реализующий интерфейс `ReactiveCrudRepository` и параметризуемый теми же обобщенными типами, что и делегирующее хранилище.
2. `ReactiveCrudRepositoryAdapter` использует базовый делегат `delegate` типа `CrudRepository`. Также адаптеру необходим экземпляр планировщика `Scheduler` для получения запросов из цикла событий. Параллелизм планировщика определяет количество одновременно обслуживаемых запросов, поэтому логично использовать то же число, что и для пула соединений. Однако прямое соответствие не всегда самое лучшее. Если пул соединений используется и для других целей, количество доступных соединений может быть меньше количества доступных потоков выполнения, и некоторые потоки могут заблокироваться в ожидании соединения (`rxjava2-jdbc` лучше обрабатывает такой сценарий).
3. Реактивный метод-обертка для блокирующего метода `save`. Блокирующий вызов заворачивается в оператор `Mono.fromCallable` (3.1) и передается выделенному планировщику `scheduler` (3.2).
4. Реактивный адаптер для метода `findById`. Сначала метод подписывается на поток идентификаторов `id` (4.1). Когда приходит очередное значение (4.2),

вызывается экземпляр делегата `delegate` (4.3). Метод `CrudRepository.findById` возвращает значение `Optional`, которое нужно отобразить в экземпляр `Mono` (4.4). В случае получения пустого значения `Optional` возвращается пустой `Mono` (4.5). Разумеется, все эти действия выполняются под управлением выделенного планировщика `scheduler`.

5. Реактивный адаптер для метода `deleteAll`. Поскольку методы `deleteAll(Publisher<T> entities)` и `deleteAll(Iterator<T> entities)` имеют разную семантику, мы не можем отобразить один реактивный вызов напрямую в один блокирующий вызов. Например, поток сущностей может быть бесконечным, следовательно, удаление элементов никогда не происходит. Поэтому метод `deleteAll` подписывается на поток сущностей (5.1) и для каждой посылает отдельный запрос `Delegate.delete (T entity)` (5.2). Поскольку запрос на удаление может выполняться параллельно, для каждого запроса выполняется свой вызов `subscribeOn`, чтобы получить рабочий поток выполнения от планировщика (5.3). Метод `deleteAll` возвращает выходной поток данных, который завершается после завершения входящего потока данных и всех операций удаления. Все методы интерфейса `ReactiveCrudRepository` должны отображаться таким образом.

Теперь определим отсутствующие методы в конкретной реализации реактивного хранилища.

```
public class RxBookRepository extends
    ReactiveCrudRepositoryAdapter<Book, Integer, BookJpaRepository> {

    public RxBookRepository(
        BookJpaRepository delegate,
        Scheduler scheduler
    ) {
        super(delegate, scheduler);
    }

    public Flux<Book> findByIdBetween(                                // (1)
        Publisher<Integer> lowerPublisher,                          //
        Publisher<Integer> upperPublisher                            //
    ) {                                                                //
        return Mono.zip(                                            // (1.1)
            Mono.from(lowerPublisher),                               //
            Mono.from(upperPublisher)                               //
        ).flatMapMany(                                             //
            function((low, upp) ->                                   // (1.2)
                Flux                                                //
                    .fromIterable(delegate.findByIdBetween(low, upp)) // (1.3)
                    .subscribeOn(scheduler)                          // (1.4)
            ))                                                       //
            .subscribeOn(scheduler);                                // (1.5)
    }
}
```

```

    public Flux<Book> findShortestTitle() {                                // (2)
        return Mono.fromCallable(delegate::findShortestTitle)           // (2.1)
            .subscribeOn(scheduler)                                       // (2.2)
            .flatMapMany(Flux::fromIterable);                             // (2.3)
    }
}

```

Класс `RxBookRepository` расширяет абстрактный класс `ReactiveCrudRepository Adapter`, ссылается на экземпляры `BookJpaRepository` и `Scheduler` и определяет следующие методы.

1. Метод `findByIdBetween` получает два реактивных потока данных и подписывается на них с помощью операции `zip` (1.1). Когда оба значения будут получены (1.2), вызывается соответствующий метод экземпляра делегата `delegate` (1.3), и выполнение блокирующего кода передается выделенному планировщику `scheduler`. Однако точно так же можно передать разрешение потоков `lowerPublisher` и `upperPublisher`, чтобы цикл обработки событий не тратил ресурсы (1.5). Но будьте осторожны, потому что в этом случае может возникнуть конкуренция за ресурсы с фактическими запросами к базе данных и снизить пропускную способность.
2. Метод `findShortestTitle` вызывает соответствующий ему метод (2.1) под управлением выделенного планировщика `scheduler` (2.2) и отображает `Iterable` в `Flux` (2.3).

Теперь наконец можно завернуть блокирующее хранилище `BookJpaRepository` в реактивное `RxBookRepository`, как показано ниже.

```

Scheduler scheduler = Schedulers.newParallel("JPA", 10);
BookJpaRepository jpaRepository = getBlockingRepository(...);

RxBookRepository rxRepository =
    new RxBookRepository(jpaRepository, scheduler);

Flux<Book> books = rxRepository
    .findByIdBetween(Mono.just(17), Mono.just(22));

books
    .subscribe(b -> log.info("Book: {}", b));

```

Не все блокирующие механизмы поддаются отображению с такой же легкостью. Например, отложенную загрузку в JPA, скорее всего, не получится адаптировать с помощью описанного подхода. Кроме того, поддержка транзакций потребует дополнительных усилий, аналогичных усилиям, предпринятым создателем библиотеки `rxjava2-jdbc`. В качестве альтернативы можно завернуть элементарные синхронные операции, чтобы ни одна транзакция не выходила за пределы одного блокирующего вызова.

Описанный здесь подход не преобразует блокирующий запрос в реактивный и неблокирующий, как по волшебству. Некоторые потоки выполнения, принадлежащие планировщику JPA, все еще будут блокироваться. Однако подробный мониторинг планировщика и разумное управление пулами могут помочь получить приемлемый баланс между производительностью приложения и использованием ресурсов.

Реактивный Spring Data в действии

Чтобы завершить эту главу и подчеркнуть преимущества реактивного доступа к базам данных, создадим реактивное приложение, обрабатывающее большие объемы данных и часто взаимодействующее с базой данных. Например, вернемся к примеру из главы 6 «Неблокирующие и асинхронные взаимодействия с *WebFlux*». Там мы реализовали альтернативное веб-приложение для службы Gitter (<https://gitter.im>). Приложение подключается к определенной комнате чата и пересылает сообщения всем подключенным пользователям через поток **серверных событий** (Server-Sent Events, SSE). Теперь, согласно новым требованиям, наше приложение должно собирать статистику о самых активных и самых популярных пользователях в чате. Наше приложение может использовать MongoDB для хранения сообщений и профилей пользователей. Эта информация также может использоваться для пересчета статистики. Диаграмма на рис. 7.18 иллюстрирует архитектуру приложения.

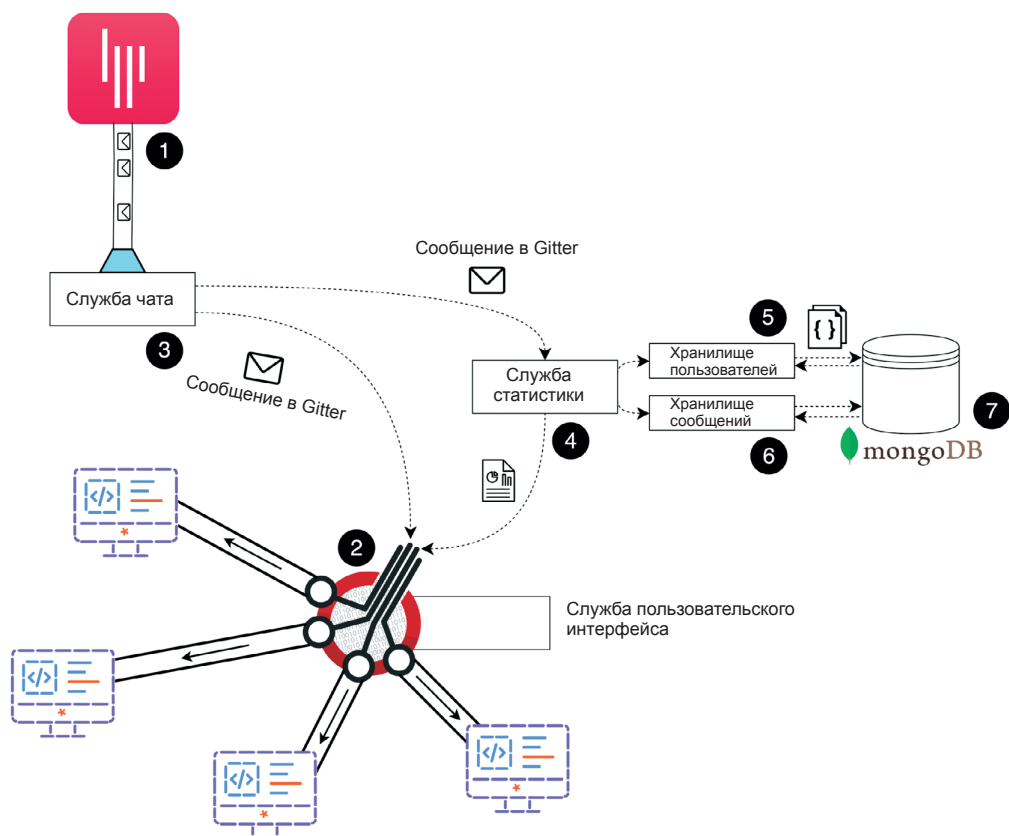


Рис 7.18. Приложение чата, использующее Spring Data для реактивного доступа к MongoDB

Рассмотрим нумерованные элементы на рис. 7.18.

1. Сервер Gitter, который может осуществлять потоковую передачу сообщений из конкретной комнаты чата через SSE. Это внешняя система, откуда приложение получает все данные.
2. **Служба пользовательского интерфейса**, которая является частью нашего приложения. Этот компонент осуществляет потоковую передачу сообщений из Gitter и последней статистики клиентам, которые являются веб-приложениями, выполняющимися в браузерах. Для потоковой передачи данных через SSE **служба пользовательского интерфейса** использует модуль WebFlux.
3. **Служба чата**, использующая реактивного веб-клиента WebClient для приема входящих сообщений от сервера Gitter. Полученные сообщения передаются в **службу пользовательского интерфейса** и в **службу статистики**.
4. **Служба статистики** непрерывно определяет самых активных и самых популярных пользователей. Статистические данные непрерывно пересылаются веб-клиентам через **службу пользовательского интерфейса**.

5. **Хранилище пользователей** – реактивное хранилище, сохраняющее информацию об участниках в MongoDB. Реализовано на основе реактивного модуля Spring Data MongoDB Reactive.
6. **Хранилище сообщений** – реактивное хранилище, позволяющее сохранять и искать сообщения из чата. Также реализовано на основе реактивного модуля Spring Data MongoDB Reactive.
7. На роль базы данных мы выбрали MongoDB, так как она хорошо соответствует требованиям приложения, имеет реактивный драйвер и хорошую поддержку в Spring Data.

Итак, поток данных в приложении постоянен и не требует каких-либо блокирующих вызовов. Служба чата пересылает сообщения через службу пользовательского интерфейса, служба статистики принимает сообщения из чата и после перерасчета отправляет статистику в службу пользовательского интерфейса. Модуль WebFlux отвечает за все сетевые взаимодействия, а Spring Data позволяет взаимодействовать с MongoDB без прерывания реактивных процессов. Здесь мы опускаем большинство деталей реализации. Однако давайте посмотрим на службу статистики.

```
public class StatisticService {
    private static final UserVM EMPTY_USER = new UserVM("", "");

    private final UserRepository userRepository;           // (1)
    private final MessageRepository messageRepository;    //

    // Конструктор...

    public Flux<UsersStatistic> updateStatistic(           // (2)
        Flux<ChatMessage> messagesFlux                   // (2.1)
    ) {
        return messagesFlux
            .map(MessageMapper::toDomainUnit)              // (2.2)
            .transform(messageRepository::saveAll)         // (2.3)
            .retryBackoff(Long.MAX_VALUE, Duration.ofMillis(500)) // (2.4)
            .onBackpressureLatest()                        // (2.5)
            .concatMap(e -> this.doGetUserStatistic(), 1)   // (2.6)
            .errorStrategyContinue((t, e) -> {});          // (2.7)
    }

    private Mono<UsersStatistic> doGetUserStatistic() {    // (3)
        Mono<UserVM> topActiveUserMono = userRepository
            .findMostActive()                             // (3.1)
            .map(UserMapper::toViewModelUnits)             // (3.2)
            .defaultIfEmpty(EMPTY_USER);                   // (3.3)

        Mono<UserVM> topMentionedUserMono = userRepository
            .findMostPopular()                             // (3.4)
    }
}
```

```

        .map(UserMapper::toViewModelUnits) // (3.5)
        .defaultIfEmpty(EMPTY_USER); // (3.6)

    return Mono.zip( // (3.7)
        topActiveUserMono,
        topMentionedUserMono,
        UsersStatistic::new
    ).timeout(Duration.ofSeconds(2)); // (3.8)
}
}

```

Проанализируем реализацию класса `StatisticService`.

1. Класс `StatisticService` имеет ссылки на `UserRepository` и `MessageRepository`, которые обеспечивают реактивную связь с коллекциями в MongoDB.
2. Метод `updateStatistic` осуществляет потоковую передачу статистических событий, представленных объектами `UsersStatistic`. Он принимает входящий поток сообщений чата, представленный аргументом `messagesFlux` (2.1). Метод подписывается на поток объектов `ChatMessage`, преобразует их в требуемое представление (2.2) и сохраняет в MongoDB с помощью `messageRepository` (2.3). Оператор `retryBackoff` помогает преодолеть возможные перебои связи с MongoDB (2.4). Также, если подписчик не может обработать все события, мы отбрасываем старые сообщения (2.5). Применяя оператор `concatMap`, запускаем процесс пересчета статистики, вызывая метод `doGetUserStatistic` (2.6). Для этого используем `concatMap`, так как он гарантирует правильный порядок результатов. Это обусловлено тем, что оператор ожидает завершения внутреннего подпотока, прежде чем генерировать следующий подпоток. Кроме того, при пересчете статистики мы игнорируем все ошибки, применяя оператор `errorStrategyContinue` (2.7), поскольку эта часть приложения не является критической и допускает некоторые временные проблемы.
3. Вспомогательный метод `doGetUserStatistic` определяет самых популярных пользователей. Для этого вызываем метод `findMostActive` в `userRepository` (3.1), преобразуем результат в правильный тип (3.2) и, если пользователь не найден, возвращаем предопределенное значение `EMPTY_USER` (3.3). Аналогично, чтобы получить самого популярного пользователя, вызываем метод `findMostPopular` хранилища (3.4), преобразуем результат (3.5) и устанавливаем значение по умолчанию, если требуется (3.6). Оператор `Mono.zip` помогает объединить эти два реактивных запроса и создать новый экземпляр класса `UsersStatistic`. Оператор `timeout` устанавливает максимальное время ожидания для пересчета статистики.

С помощью этого элегантного кода мы легко смешали входящий поток сообщений от объекта `WebClient` и исходящий поток событий SSE, обрабатываемых модулем `WebFlux`. Естественно, мы также включили в наш реактивный конвейер об-

работку запросов MongoDB, воспользовавшись реактивной реализацией Spring Data. Более того, мы нигде не заблокировали никаких потоков выполнения. В результате наше приложение очень эффективно использует ресурсы сервера.

Заключение

В этой главе мы узнали много нового об организации хранения данных в современных приложениях. Описали проблемы доступа к данным в рамках микросервисной архитектуры, рассказали, как использование хранилищ разного типа помогает создавать службы с желаемыми характеристиками. Рассмотрели доступные варианты реализации распределенных транзакций. Эта глава поведала о плюсах и минусах блокирующих и реактивных подходов к хранению данных, а также об отсутствующих реактивных альтернативах для каждого из современных блокирующих слоев доступа к данным.

Мы рассказали, как изящно проект Spring Data обеспечивает реактивный доступ к данным в современных приложениях Spring. Исследовали особенности реализации реактивных коннекторов для MongoDB и Cassandra. Рассмотрели поддержку многодокументных транзакций в MongoDB 4. В этой главе описаны доступные версии реактивных API следующего поколения на уровне языка для доступа к базам данных – ADBA и R2DBC. Мы изучили достоинства и недостатки обоих подходов и узнали, как Spring Data может поддерживать реактивные хранилища для реляционных баз данных с новым модулем Spring Data JDBC.

Кроме того, мы рассмотрели текущие варианты интеграции блокирующих драйверов или хранилищ с реактивным приложением и узнали много нового! Но мы лишь слегка затронули тему хранения данных, поскольку ее невозможно рассмотреть целиком в одной главе.

В начале главы мы упомянули двойственную природу баз данных – хранение статических данных и потоков сообщений с обновлениями данных. В следующей главе рассмотрим реактивные системы и реактивное программирование в контексте систем обмена сообщениями, таких как Kafka и RabbitMQ.

Глава 8

Масштабирование с Cloud Streams

В предыдущих главах рассказывалось, как благодаря Reactor 3 работа с парадигмой реактивного программирования может стать удовольствием. Теперь мы знаем, как получить реактивное веб-приложение с использованием Spring WebFlux и Spring Data Reactive. Эта надежная комбинация позволяет создать приложение, способное справиться с высокой нагрузкой, а также обеспечивает эффективное использование ресурсов, низкое потребление памяти, низкую задержку и высокую пропускную способность.

Однако этим возможности экосистемы Spring не ограничиваются. В данной главе мы узнаем, как улучшить приложение, используя функции, предлагаемые экосистемой Spring Cloud, а также выясним, как построить полностью реактивную систему с использованием **Spring Cloud Streams**. Кроме того, мы познакомимся с библиотекой RSocket и узнаем, чем она может помочь нам в разработке систем быстрой потоковой передачи данных. Наконец, рассмотрим модуль **Spring Cloud Function**, упрощающий создание облачной реактивной системы с использованием приемов реактивного программирования и с поддержкой обратного давления.

Будут рассмотрены следующие темы:

- роль брокеров сообщений в реактивных системах;
- роль Spring Cloud Streams в реактивных системах на основе Spring Framework;
- реализация бессерверных реактивных систем с Spring Cloud Function;

- RSocket как прикладной протокол для асинхронной передачи сообщений с низкой задержкой.

Брокеры сообщений как основа систем, управляемых сообщениями

Как рассказывалось в главе 1 «Причины выбора Spring», суть реактивных систем заключается в обмене сообщениями. Более того, из предыдущих глав мы узнали, что методы реактивного программирования позволяют реализовать асинхронные взаимодействия между процессами/службами. Кроме того, опираясь на стандарт Reactive Streams, можно асинхронно управлять обратным давлением и отказами. Собрав все это воедино, можно создать распределенное реактивное приложение, действующее на одном компьютере. К сожалению, такие приложения, выполняющиеся на одном узле, имеют свои ограничения, обусловленные возможностями аппаратуры. Прежде всего невозможно предоставить новые вычислительные ресурсы, такие как дополнительный процессор, ОЗУ и жесткий диск/твердотельный накопитель, не отключая всю систему. Подобное решение не приносит никакой пользы, поскольку пользователи могут быть распределены по всему миру и эффективность их обслуживания может отличаться.



Здесь под эффективностью обслуживания пользователей подразумевается разное распределение задержки, которое напрямую зависит от расстояния между местоположением сервера приложения и местоположением пользователя.

Такие ограничения можно устранить, разделив монолитное приложение на микросервисы. Цель метода – получить эластичную систему с простой прозрачностью местоположения. Однако при таком способе появляются новые проблемы, такие как управление службами, мониторинг и безболезненное масштабирование.

Балансировка нагрузки на стороне сервера

На самых ранних этапах развития распределенных систем одним из способов достижения прозрачности местоположения и эластичности было использование внешних балансировщиков нагрузки, таких как HAProxy/Nginx, в качестве точки входа поверх группы реплик или в качестве центрального балансировщика нагрузки для всей системы. Взгляните на диаграмму, изображенную на рис. 8.1.

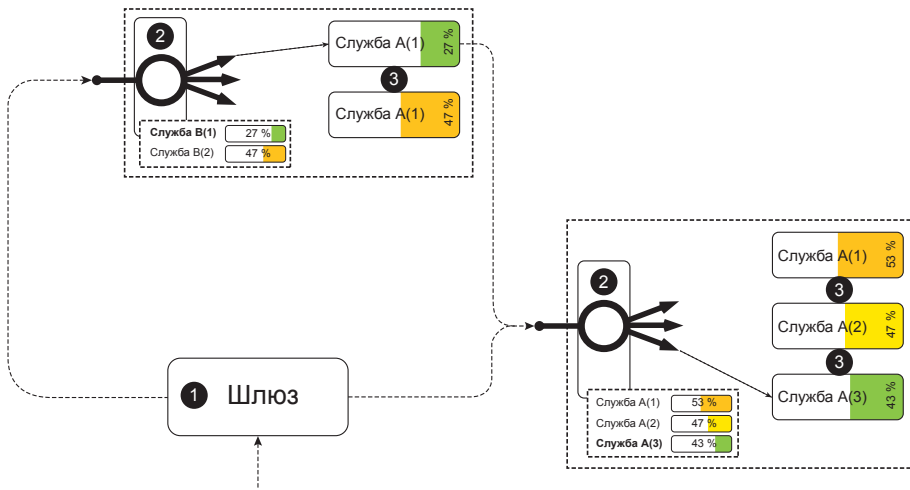


Рис. 8.1. Пример балансирования нагрузки с использованием внешней службы

Рассмотрим нумерованные элементы на рис. 8.1.

1. Здесь имеем службу, которая играет роль шлюза и координирует все запросы пользователей. Шлюз выполняет два вызова: к **службе А** и **службе В**. Предположим, **служба А** играет роль точки доступа к шлюзу, проверяя правильность ключа доступа или выполняя авторизацию при первом обращении. После проверки прав доступа выполняется второй вызов, запускающий выполнить бизнес-логику в **службе В**, для чего может понадобиться дополнительная проверка разрешений. В результате снова вызывается служба контроля доступа.
2. Схематическое представление балансировщика нагрузки. Для поддержки автоматического масштабирования балансировщик нагрузки может учесть такой показатель, как количество открытых соединений, служащий оценкой общей нагрузки на службу. В качестве альтернативы балансировщик нагрузки может определять задержки ответа и на их основе делать некоторые дополнительные предположения о работоспособности службы. Комбинируя эту информацию с периодической проверкой работоспособности, балансировщик нагрузки может вызывать сторонний механизм для выделения дополнительных ресурсов на пиках нагрузки или освобождать избыточные узлы при снижении нагрузки.
3. Конкретные экземпляры служб, сгруппированные по балансировщику нагрузки (2). Каждая служба может работать независимо на отдельных узлах или машинах.

Как показано на рис. 8.1, балансировщик нагрузки играет роль реестра доступных экземпляров. Для каждой группы служб имеется выделенный балансировщик, который распределяет нагрузку между экземплярами. Балансировщик на-

грузки, в свою очередь, инициирует процесс масштабирования на основе общей групповой нагрузки и имеющихся показателей. Например, когда наблюдается всплеск активности пользователей, в группу могут динамически добавляться новые экземпляры для обработки повышенной нагрузки. Когда нагрузка уменьшается, балансировщик (как держатель показателей) отправляет уведомление о том, что в группе есть избыточные экземпляры.



Обратите внимание, что в книге мы не будем рассматривать методы автоматического масштабирования, но в главе 10 *«И наконец, выпуск!»* расскажем о функциях мониторинга и проверки работоспособности.

Однако у этого решения есть несколько проблем. Прежде всего при высокой нагрузке балансировщик может стать горячей точкой в системе. Согласно закону Амдала, балансировщик нагрузки становится узким местом, из-за чего находящаяся за ним группа служб не в состоянии обработать больше запросов, чем сможет пропустить балансировщик нагрузки. Кроме того, стоимость обслуживания балансировщика нагрузки может быть очень высокой, поскольку для каждого выделенного балансировщика нагрузки требуется отдельная мощная физическая или виртуальная машина. Также может потребоваться дополнительная резервная машина с установленным балансировщиком. Наконец, балансировщик нагрузки также должен управляться и контролироваться. Это может привести к дополнительным расходам на администрирование инфраструктуры.



Вообще говоря, балансировка нагрузки на стороне сервера – это метод, проверенный временем, который можно использовать во многих случаях. Да, он имеет свои ограничения, но в настоящее время является вполне надежным решением. Узнать больше о приемах и вариантах использования балансировки нагрузки можно по ссылке <https://aws.amazon.com/ru/blogs/devops/introducing-application-load-balancer-unlocking-and-optimizing-architectures/>.

Балансировка нагрузки на стороне клиента с Spring Cloud и Ribbon

К счастью, экосистема Spring Cloud помогает решить проблему, когда балансировщик нагрузки на стороне сервера становится горячей точкой в системе. Вместо того чтобы предоставить обходной путь для внешнего балансировщика нагрузки, команда Spring решила воспользоваться опытом Netflix для создания распределенных систем. Один из способов достижения масштабируемости и прозрачности местоположения заключается в балансировке нагрузки на стороне клиента.

Идея балансировки нагрузки на стороне клиента проста и предполагает взаимодействие со службой через сложного клиента, который знает о доступных экземп-

лярах службы и может легко сбалансировать нагрузку между ними, как показано на рис. 8.2.

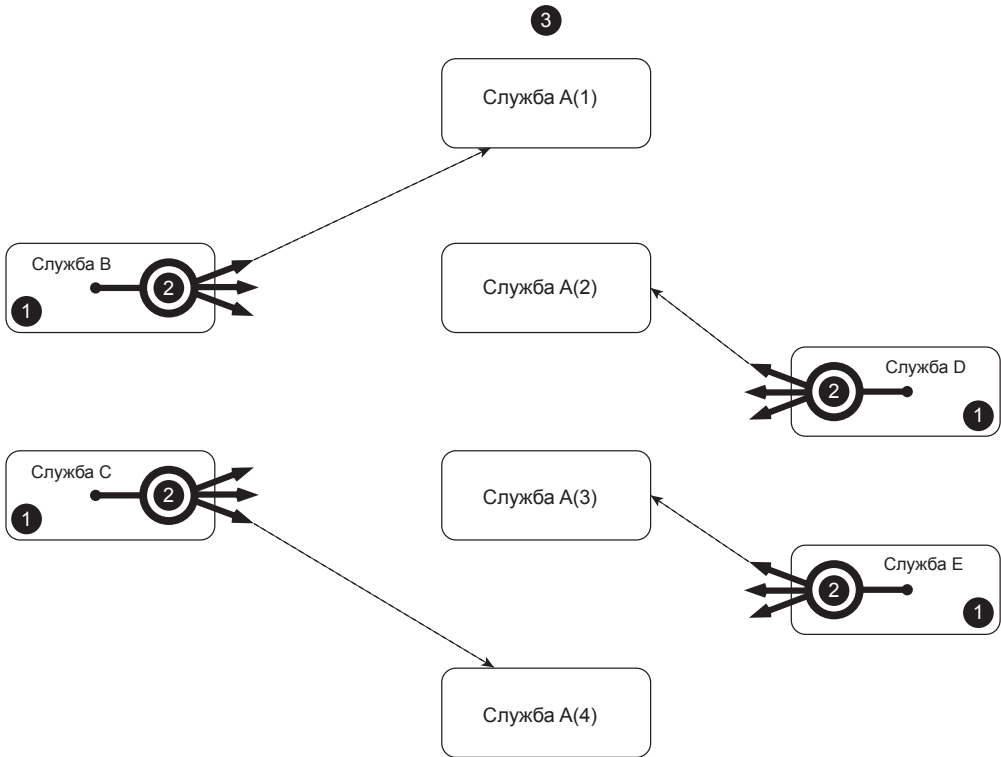


Рис. 8.2. Пример реализации балансировки на стороне клиента

Рассмотрим нумерованные элементы на рис. 8.2.

1. Здесь мы имеем несколько служб, взаимодействующих со **службой А**.
2. Балансировщики на стороне клиента. Теперь балансировщик является частью каждой службы. Соответственно, вся координация осуществляется до отправки фактического HTTP-запроса.
3. Группа экземпляров **службы А**.

В этом примере все вызывающие службы обращаются к разным репликам **службы А**. Несмотря на то что метод обеспечивает независимость от выделенного внешнего балансировщика нагрузки (и лучшую масштабируемость), у него есть ограничения. Прежде всего данный способ балансировки нагрузки – это балансировка на стороне клиента. Следовательно, каждый клиент отвечает за балансировку всех запросов локально, выбирая экземпляр целевой службы, что помогает избежать единой точки отказа, тем самым обеспечивая лучшую масштабируемость. С другой стороны, информация о доступных службах должна быть каким-то образом доступна остальным службам в системе.

Чтобы ознакомиться с современными методами обнаружения служб, рассмотрим одну из популярных библиотек в экосистеме Java и Spring – Ribbon. Библиотека Ribbon реализует шаблон балансировки нагрузки на стороне клиента, придуманный в Netflix. Ribbon предлагает два распространенных способа доступа к списку доступных служб. Простейшим из них является заранее подготовленный статический список адресов служб. Этот прием изображен на рис. 8.3.

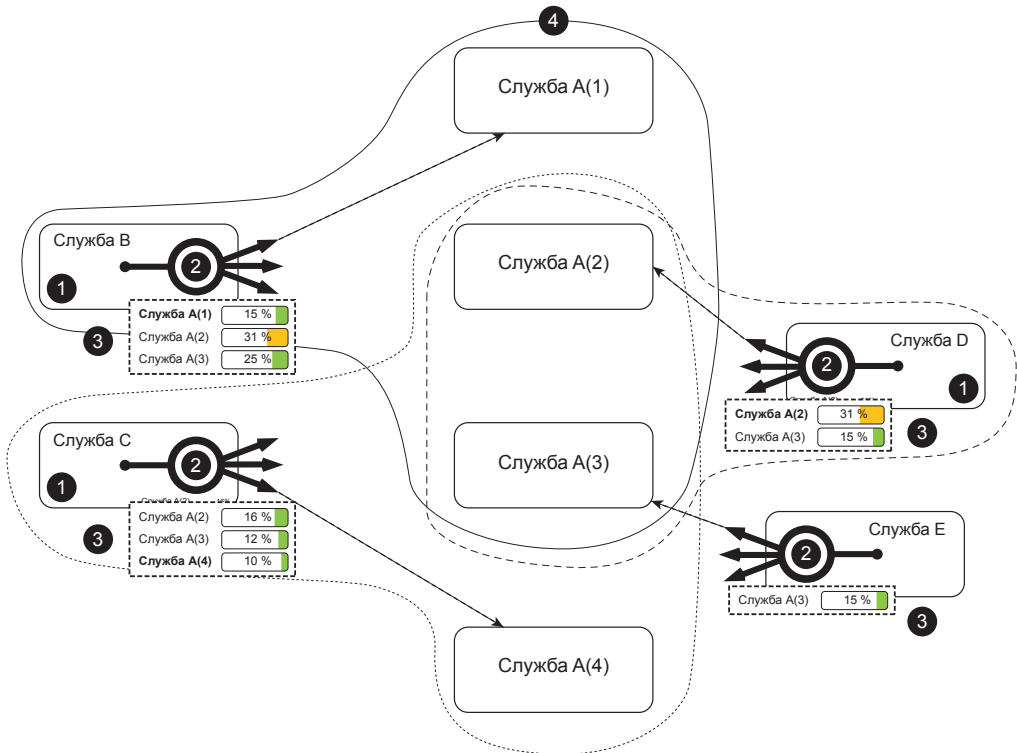


Рис. 8.3. Заранее подготовленный статический список служб для балансировки нагрузки на стороне клиента с Netflix Ribbon

Рассмотрим нумерованные элементы на рис. 8.3.

1. Здесь имеется несколько служб, взаимодействующих со **службой А**.
2. Балансировщики нагрузки, действующие на стороне клиента. Каждый имеет доступ к определенной группе экземпляров служб.
3. Представление внутреннего списка предварительно настроенных экземпляров **службы А**. Каждый вызывающий независимо оценивает нагрузку на каждый целевой экземпляр **службы А** и применяет процедуру балансировки. Имя службы, выделенное жирным шрифтом в списке, обозначает текущий целевой экземпляр **службы А**.
4. Фактическая группа экземпляров **службы А**.

На рис. 8.3 фигуры с разными границами обозначают области знаний разных клиентов. К сожалению, методика балансировки нагрузки на стороне клиента имеет свои недостатки. Прежде всего отсутствует координация между балансировщиками, поэтому может получиться так, что все клиенты решат вызвать один и тот же экземпляр и вызвать перегрузку, как показано на рис. 8.4

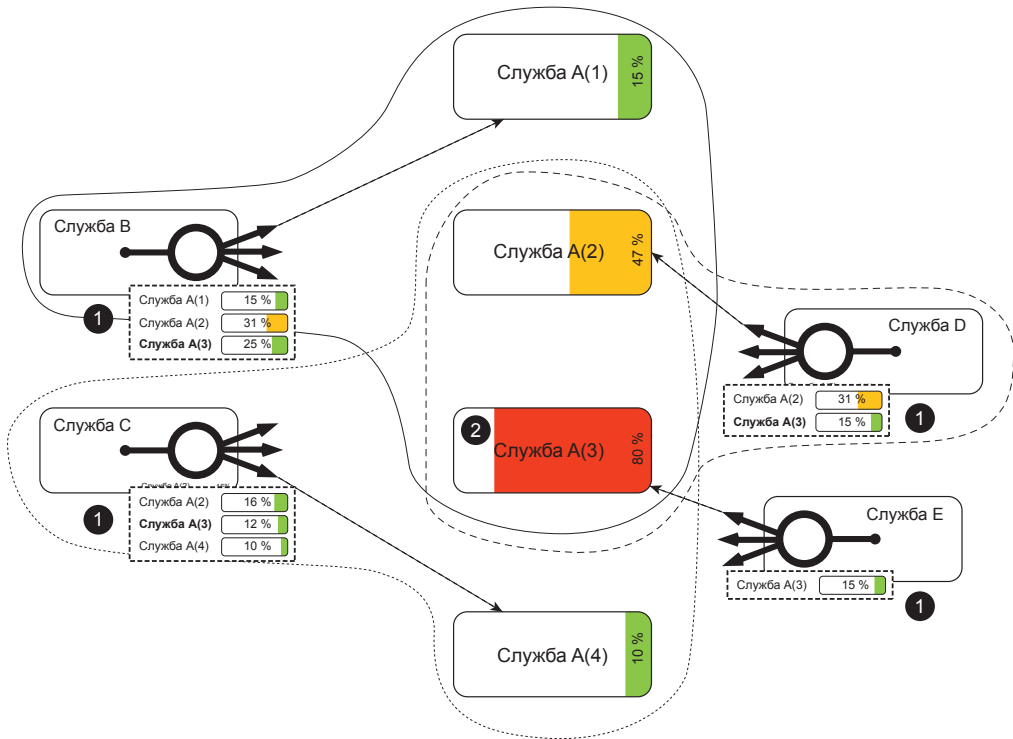


Рис. 8.4. Проблема, вызванная отсутствием координации между балансировщиками на стороне клиента

Рассмотрим нумерованные элементы на рис. 8.4.

1. Вызывающие службы со списками предварительно настроенных экземпляров **службы А**. Локальные оценки нагрузки, разные для разных служб. В этом примере показан случай, когда все службы обращаются к одному и тому же целевому экземпляру **службы А**. Такое поведение может вызвать неожиданный скачок нагрузки.
2. Один из экземпляров **службы А**. Фактическая нагрузка на экземпляр отличается от предполагаемой каждой вызывающей службой.

Кроме того, такой простой способ управления нагрузкой с помощью статического списка экземпляров службы далек от требований реактивных систем, главным образом с точки зрения эластичности к нагрузке.



В манифесте реактивных систем под *эластичностью* понимается способность системы динамически увеличивать пропускную способность в ответ на растущий спрос и снижать потребление ресурсов с уменьшением спроса.

Для решения этой проблемы библиотека Ribbon поддерживает интеграцию с реестром служб, таким как Eureka, который непрерывно обновляет список доступных реплик службы. Взгляните на рис. 8.5.

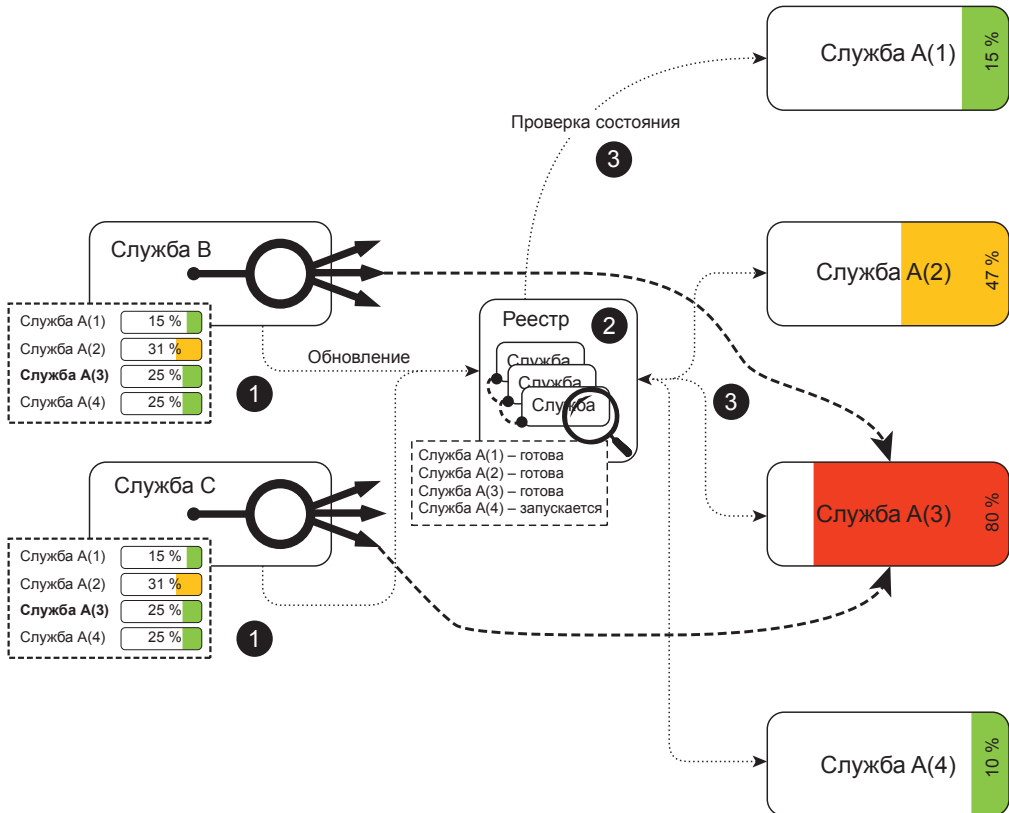


Рис. 8.5. Пример динамически обновляемого списка служб

Рассмотрим нумерованные элементы на рис. 8.5.

1. Вызывающие службы со списками доступных экземпляров **службы А**. Для поддержания в актуальном состоянии списка действующих экземпляров **службы А** клиентский балансировщик периодически обновляет этот список и извлекает из реестра самую последнюю информацию (2).
2. Представление реестра служб. Реестр хранит свой список служб с их состоянием.
3. Пунктирные линии представляют запросы для получения состояния служб.

Как показано на рис. 8.5, проблема координации клиентских балансировщиков нагрузки остается. В этом случае реестр отвечает за хранение списка действующих экземпляров службы и его постоянное обновление во время выполнения. Балансировщик на стороне клиента и реестр могут хранить информацию о нагрузке экземпляров целевой службы, и балансировщик может периодически синхронизировать свою информацию о нагрузке с информацией, собираемой реестром. Кроме того, все заинтересованные стороны могут получить доступ к этой информации и выполнить балансировку нагрузки на ее основе. Данный способ управления списком доступных служб намного шире предыдущего и позволяет динамически обновлять список доступных экземпляров.

Метод хорошо работает для небольшого кластера служб. Однако динамическое обнаружение служб с использованием общего реестра далеко от идеала. Как и в случае с балансировкой нагрузки на стороне сервера, классический реестр служб, такой как Eureka, становится единственной точкой отказа и требует огромных усилий для поддержания в актуальном состоянии сведений о состоянии системы. Например, если состояние кластера быстро меняется, информация о зарегистрированных службах может устаревать. Для слежения за работоспособностью экземпляры службы обычно периодически отправляют служебные сообщения. Как вариант реестр сам может периодически посылать запросы для проверки работоспособности экземпляров. В обоих случаях очень частые обновления состояния могут потреблять неоправданно высокий процент ресурсов кластера. Поэтому продолжительность между проверками обычно длится от нескольких секунд до нескольких минут (по умолчанию в Eureka интервал проверки составляет 30 секунд). Следовательно, реестр может предложить экземпляр службы, который был исправен во время последней проверки, но к данному моменту уже уничтожен. Следовательно, чем динамичнее кластер, тем сложнее точно отслеживать состояние служб с помощью централизованного реестра.

Кроме того, балансировка все еще происходит на стороне клиента. Возникает та же проблема несогласованности, а значит, есть вероятность, что фактическая нагрузка на службу окажется несбалансированной. Более того, точная и честная координация запросов в распределенной системе, обеспечиваемая балансировщиками нагрузки на стороне клиента, является еще одной проблемой, которая, вероятно, сложнее предыдущей. Следовательно, мы должны найти лучшее решение для построения реактивной системы.



В этом разделе мы обсудили очень популярную в экосистеме Spring Cloud службу реестра Eureka. За дополнительной информацией обращайтесь по ссылке <https://cloud.spring.io/spring-cloud-netflix/>. В целом балансировка нагрузки на стороне клиента является эффективным методом распределения нагрузки между целевыми экземплярами службы.

Кроме того, существуют алгоритмы, обеспечивающие предсказуемость балансировки на стороне клиента, поэтому большинства описанных здесь проблем можно избежать. Чтобы узнать больше, обращайтесь по ссылке <https://www.youtube.com/watch?v=6NdXUY1La2I>.

Брокеры сообщений как эластичный и надежный слой для передачи сообщений

К счастью, манифест реактивных систем предлагает решение проблем, связанных с балансировкой нагрузки на стороне сервера и на стороне клиента:

«Использование явной передачи сообщений позволяет управлять нагрузкой, эластичностью и потоком управления за счет формирования очереди сообщений и управления ею в системе и применения обратного давления при необходимости».

Это утверждение можно истолковать как рекомендацию использовать независимые брокеры сообщений для передачи сообщений. Взгляните на рис. 8.6.

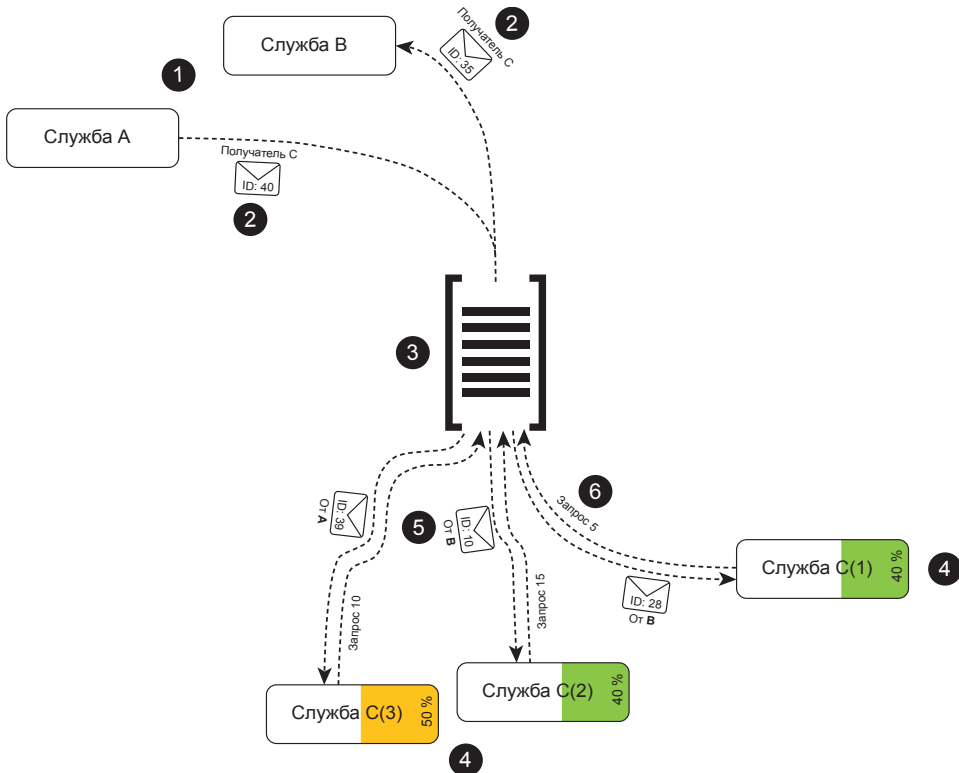


Рис. 8.6. Пример балансировки нагрузки с применением очереди сообщений

Рассмотрим нумерованные элементы на рис. 8.6.

1. Вызывающие службы. Они знают только местоположение очереди сообщений и имя службы-получателя, что позволяет отделить вызывающую службу от целевого экземпляра. Эта модель взаимодействий подобна той, что используется для балансировки на стороне сервера. Одно из существенных отличий заключается в асинхронном способе общения между вызывающей службой и конечным получателем. В этом случае нам не требуется удерживать соединение открытым на протяжении всего времени обработки запроса.
2. Представление исходящего сообщения. Исходящее сообщение может содержать информацию с именем службы-получателя и идентификатором сообщения.
3. Представление очереди сообщений. Очередь сообщений работает как независимая служба и позволяет пользователям отправлять сообщения группе экземпляров **службы С**, чтобы любой доступный экземпляр мог обрабатывать сообщения.
4. Экземпляры службы получателя. Каждый экземпляр **службы С** имеет одинаковую среднюю нагрузку, потому что все они реализуют управление посредством обратного давления, сообщая о готовности (6), чтобы очередь могла отправить следующее входящее сообщение (5).

Все запросы отправляются через очередь сообщений, которая затем может передать их любому свободному экземпляру. Кроме того, очередь сообщений может хранить сообщение, пока один из экземпляров не запросит новых сообщений. При такой организации очередь сообщений знает, сколько в системе заинтересованных сторон, и может управлять нагрузкой, исходя из этой информации. Каждый экземпляр, в свою очередь, может самостоятельно управлять обратным давлением и посылать запросы на получение новых сообщений в зависимости от возможностей машины. Просто наблюдая за количеством ожидающих сообщений, мы можем увеличить количество активных экземпляров для их обработки. Кроме того, наблюдая за количеством простаивающих экземпляров, можем останавливать избыточные экземпляры.

Хотя очередь сообщений решает проблему балансировки нагрузки на стороне клиента, может показаться, что мы возвращаемся к очень похожему решению балансировки нагрузки на стороне сервера, которое рассмотрено выше, и очередь сообщений может стать горячей точкой в системе. Однако это не так. Прежде всего сама модель взаимодействий здесь немного другая. Вместо поиска доступных служб и выбора, кому отправить запрос, очередь сообщений просто помещает входящее сообщение в очередь. Затем, когда экземпляр службы-обработчика объявит о намерении получить сообщения, они будут отправлены из очереди. Соответственно, мы имеем два отдельных и, возможно, независимых этапа:

- прием сообщений и помещение их в очередь (что выполняется очень быстро);

- передача данных, когда потребитель заявит о своей потребности.

С другой стороны, можем создать отдельный экземпляр очереди сообщений для каждой группы получателей и тем самым улучшить масштабируемость системы и благополучно избежать любых узких мест. Взгляните на рис. 8.7.

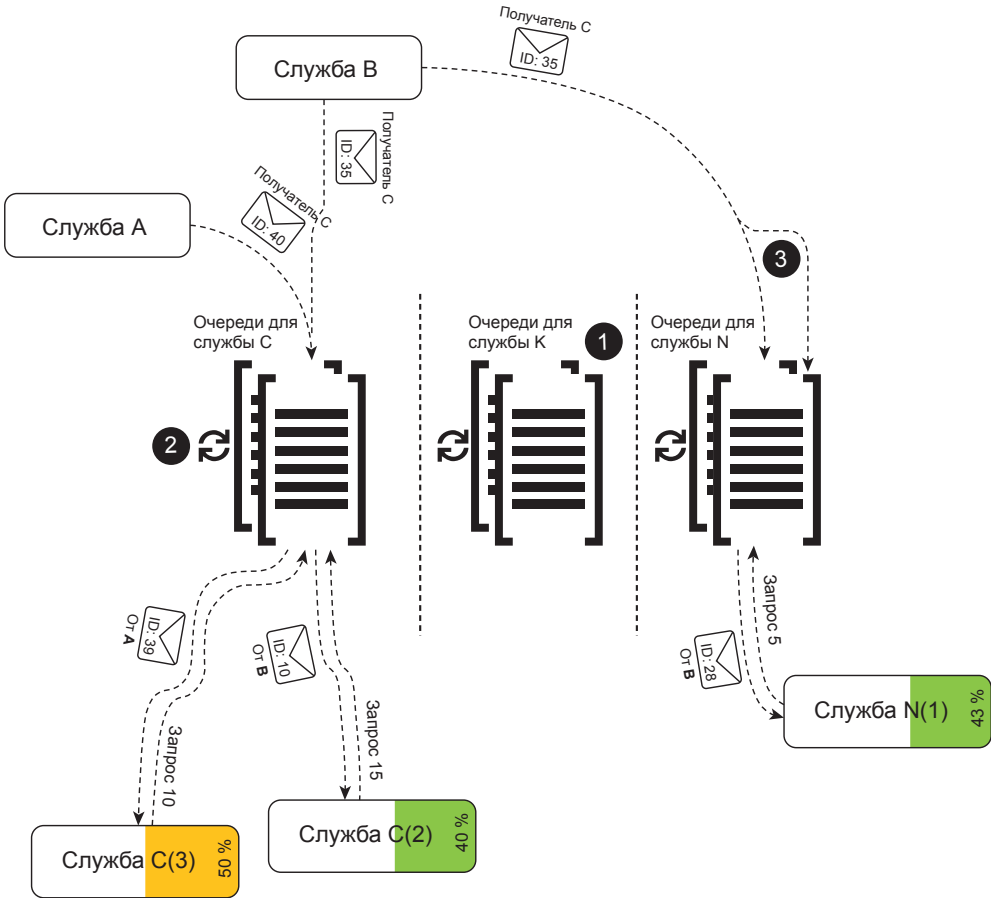


Рис. 8.7. Пример увеличения эластичности с очередями сообщений в виде служб

Рассмотрим нумерованные элементы на рис. 8.7.

1. Очереди сообщений с поддержкой репликации. В данном примере имеется несколько таких очередей сообщений, по одной для каждой группы экземпляров службы.
2. Синхронизация состояния между репликами для одной и той же группы получателей.
3. Представление балансировки нагрузки (например, на стороне клиента) между репликами.

Здесь есть отдельная очередь для каждой группы получателей и набор реплик для каждой очереди в группе. Однако нагрузка может меняться от группы к группе, поэтому одна может быть перегружена, а другая – простаивать без работы, впуская расходы ресурсы. Следовательно, вместо создания очереди сообщений как службы мы могли бы создать брокера сообщений, поддерживающего виртуальные очереди. Используя этот подход, можем снизить стоимость инфраструктуры, поскольку нагрузка на систему снизится и единственный брокер сообщений будет способен обслужить сразу несколько групп получателей. Брокер сообщений, в свою очередь, тоже может быть реактивной системой, а значит, гибким, отказоустойчивым и поддерживающим асинхронную неблокирующую передачу сообщений. Взгляните на рис. 8.8.

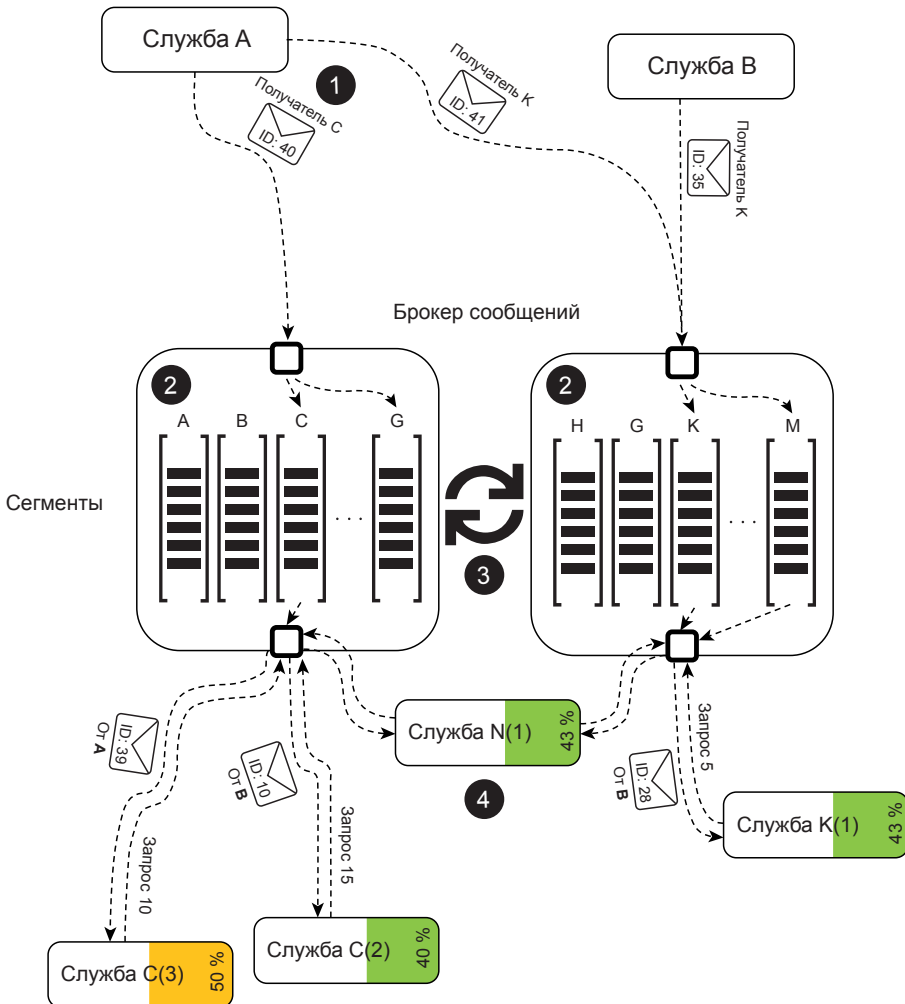


Рис. 8.8. Эластичность с распределенным брокером сообщений

Рассмотрим нумерованные элементы на рис. 8.7.

1. Вызывающая служба с сегментированным балансировщиком нагрузки на стороне клиента. В общем случае брокер сообщений может использовать приемы, описанные выше, для обнаружения сегментов и передачи информации клиентам.
2. Представление сегмента брокера сообщений. Каждый сегмент имеет несколько привязанных к нему получателей (тем). Кроме того, каждый сегмент также может иметь свои реплики.
3. Перебалансировка сегмента. Брокер сообщений может использовать дополнительный механизм перебалансировки, чтобы с появлением новых получателей или узлов в кластере брокер сообщений мог легко масштабироваться.
4. Пример получателя, который может принимать сообщения из разных сегментов.

На рис. 8.8 изображен возможный дизайн брокера сообщений как системы, которая может стать надежной основой для целевого приложения. Как показано на рис. 8.8, брокер сообщений может содержать столько виртуальных очередей, сколько требуется системе. Современные брокеры сообщений применяют такие методы, как согласованность в конечном итоге и многоадресная рассылка сообщений, что позволяет добиться гибкости из коробки. Брокеры сообщений могут играть роль надежного слоя для асинхронной передачи сообщений с поддержкой обратного давления и гарантией воспроизводимости.



Например, надежность брокера сообщений можно увеличить, используя эффективный метод репликации и сохранения сообщений в быстрых хранилищах. Однако производительность таких брокеров сообщений может быть ниже производительности других брокеров сообщений, которые не применяют долговременных хранилищ для хранения сообщений или когда сообщения пересылаются напрямую между узлами.

Что это означает для нас с вами? А то, что в случае сбоя брокера сообщений все сообщения останутся доступными, следовательно, как только восстановится слой обмена сообщениями, все недоставленные сообщения найдут своих адресатов.

Итак, можно отметить, что применение брокера сообщений улучшает общую масштабируемость системы. В этом случае можем легко создать эластичную систему только потому, что брокер сообщений способен действовать подобно реактивной системе. В результате взаимодействия перестают быть узким местом.

Рынок брокеров сообщений

Несмотря на то что в большинстве случаев идея использования брокеров сообщений может показаться идеальным решением, фактическая реализация собственного брокера сообщений сродни кошмару. К счастью, сегодня рынок предлага-

ет несколько мощных решений с открытым исходным кодом. К числу наиболее популярных брокеров сообщений и платформ обмена сообщениями относятся RabbitMQ, Apache Kafka, Apache Pulsar, Apache RocketMQ, NATS и др.



Сравнение брокеров сообщений можно найти по следующим ссылкам: Apache RocketMQ и Apache Kafka по адресу <https://rocketmq.apache.org/docs/motivation/#rocketmq-vs-activemq-vs-kafka>; RabbitMQ, Kafka и NSQ – <https://stackshare.io/stackups/kafka-vs-nsq-vs-rabbitmq>; Apache Pulsar и Apache Kafka – <https://streaml.io/blog/pulsar-streaming-queuing>.

Spring Cloud Streams как мост в экосистему Spring

Каждое из решений, упомянутых выше, имеет свои преимущества, такие как низкая задержка или более надежные гарантии доставки либо сохранности сообщений.

Однако книга рассказывает о реактивных возможностях в экосистеме Spring, поэтому полезно получить представление о том, что предлагает Spring для гладкой интеграции с брокерами сообщений.

Spring Cloud Streams предлагает один из мощных способов создания надежных систем, управляемых сообщениями, с использованием Spring Cloud. Spring Cloud Streams предоставляет упрощенную модель программирования для асинхронного обмена сообщениями между службами. Модуль Spring Cloud Streams, в свою очередь, реализован поверх модулей Spring Integration и Spring Message, которые являются фундаментальными абстракциями интеграции с внешними службами и прямой асинхронной передачи сообщений. Более того, Spring Cloud Streams дает возможность создавать эластичные приложения без необходимости иметь дело со слишком сложными конфигурациями и не требует глубоких знаний конкретного брокера сообщений.



К сожалению, круг брокеров сообщений, имеющих соответствующую поддержку в Spring Framework, очень ограничен. На момент написания этих строк Spring Cloud Streams поддерживал интеграцию только с RabbitMQ и Apache Kafka.

Чтобы разобраться в основах построения реактивных систем с помощью Spring Cloud Streams, пересмотрим дизайн реактивного приложения чата, которое создали в главе 7 «Реактивный доступ к базам данных», используя новый реактивный стек Spring Cloud.

Начнем с краткого обзора дизайна нашего приложения, состоящего из трех концептуальных частей. Первая часть – это служба коннекторов ChatService. В на-

шем случае она реализует связь со службой Gitter, которая представляет поток событий, отправляемых сервером. Поток сообщений распределяется между службой ChatController, отвечающей за передачу сообщений непосредственно конечным пользователям, и службой StatisticService, отвечающей за хранение сообщений в базе данных и пересчет статистики. Прежде все три части входили в состав одного монолитного приложения. Соответственно, каждый компонент в системе подключался с помощью механизма внедрения зависимостей в Spring Framework. Кроме того, асинхронный и неблокирующий обмен сообщениями поддерживался реактивными типами из Reactor 3. Первое, что нам нужно понять, – позволяет ли Spring Cloud Streams разбить монолитное приложение на микросервисы и использовать реактивные типы для связи между компонентами.

К счастью, начиная с версии Spring Cloud 2, существует прямая поддержка взаимодействий с использованием типов из Reactor. Ранее прозрачность местоположения могла зависеть от слабой связанности компонентов в монолитном приложении. Используя **инверсию управления** (Inversion of Control, IoC), каждый компонент может получить доступ к интерфейсу другого компонента, не имея ни малейших представлений о реализации. В облачной экосистеме наряду со знанием интерфейса доступа мы должны знать имя домена (имя компонента) или (в нашем случае) имя выделенной очереди. В качестве замены взаимодействий через интерфейсы Spring Cloud Streams предлагает две концептуальные аннотации для связи между службами.



Чтобы узнать больше о понятии «прозрачность местоположения (Location Transparency)», обращайтесь по ссылке <http://wiki.c2.com/?LocationTransparency>.

Первая концептуальная аннотация – аннотация `@Output`. Она определяет имя очереди, в которую должно быть доставлено сообщение. Вторая концептуальная аннотация – аннотация `@Input`, которая определяет очередь, из которой должны извлекаться сообщения. Данный метод взаимодействий между службами может заменить интерфейсы, поэтому вместо вызова метода можем положиться на отправку сообщений в определенную очередь. Давайте рассмотрим изменения, которые нужно внести в приложение, чтобы организовать отправку сообщений брокеру сообщений.

```
@SpringBootApplication                                // (1)
@EnableBinding(Source.class)                          // (1.1)
@EnableConfigurationProperties(...)                    //
/* @Service */                                           // (1.2)
public class GitterService                               //
    implements ChatService<MessageResponse> {           //
    ...                                                  // (2)
@StreamEmitter                                         // (3)
```

```

@Output(Source.OUTPUT)                                     // (3.1)
public Flux<MessageResponse> getMessagesStream() { ... }    //

@StreamEmitter                                             // (4)
@Output(Source.OUTPUT)                                     //
public Flux<MessageResponse> getLatestMessages() { ... }    //

public static void main(String[] args) {                  // (5)
    SpringApplication.run(GitterService.class, args);        //
}                                                            //
}                                                            //

```



Обратите внимание, что вместе с фактической реализацией в предыдущем листинге показаны отличия. Комментарии вида `/*` Закомментированный код `*/` обозначают удаленные строки, а жирным шрифтом выделены новые строки. Код, набранный обычным шрифтом, – это неизменившиеся строки.

Пояснения к коду.

1. Объявление `SpringBootApplication`. В строке (1.1) мы используем аннотацию `@EnableBinding`, которая обеспечивает базовую интеграцию с потоковой инфраструктурой (например, с Apache Kafka). Мы также удалили аннотацию `@Service` (1.2), поскольку мигрировали из монолитного приложения в распределенное. Теперь можем запустить этот компонент как небольшое независимое приложение и тем самым обеспечить лучшее масштабирование.
2. Список полей и конструкторов, которые не изменились.
3. Относится к объявлению потока `Flux` сообщений. Метод возвращает бесконечный поток сообщений от Gitter. Ключевую роль здесь играет аннотация `@StreamEmitter`, которая объявляет данный источник реактивным. Поэтому, чтобы определить канал назначения, используется аннотация `@Output` с именем канала. Обратите внимание, что имя целевого канала должно присутствовать в списке связанного канала в строке (1.1).
4. Метод `getLatestMessages` возвращает конечный поток с последними сообщениями из Gitter и посылает их в канал назначения.
5. Объявление метода `main`, который осуществляет запуск приложения Spring Boot.

Как показано в примере, существенных изменений с точки зрения бизнес-логики не произошло. С другой стороны, простым применением нескольких аннотаций Spring Cloud Streams здесь добавилась обширная инфраструктурная логика. Прежде всего, используя `@SpringBootApplication`, мы определили нашу маленькую службу как отдельное приложение Spring Boot. С помощью `@Output(Source.OUTPUT)` определили имя очереди назначения в брокере сообщений.

Наконец, добавили аннотации `@EnableBinding` и `@StreamEmitter`, связав наше приложение с брокером сообщений, и методы `getMessagesStream()` и `getLatestMessages()`, которые вызываются в начальной точке приложения.



Узнать больше о `@StreamEmitter` и ее ограничениях можно по ссылке: https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_reactive_sources. Также получить упрощенное представление о модели аннотаций в Spring Cloud Stream можно по ссылке https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_programming_model.

В дополнение к Java-аннотациям мы должны определить конфигурацию привязки к Spring Cloud Stream. Это можно сделать путем предоставления свойств Spring Application, как показано ниже (в данном случае `application.yaml`).

```
spring.cloud.stream:                               //
  bindings:                                         //
    output:                                         // (1)
      destination: Messages                        // (2)
      producer:                                     // (3)
        requiredGroups: statistic, ui              // (4)
```

В данном примере в строке (1) мы указали ключ привязки, который является именем канала, определенного в `Source.OUTPUT`. Благодаря этому можем получить доступ к `org.springframework.cloud.stream.config.BindingProperties` и настроить имя получателя в брокере сообщений (2). Наряду с этим можно настроить поведение нашего производителя (3). Например, можно настроить список получателей, которые должны получать сообщения с гарантией доставки *хотя бы один раз* (4).

Запустив предыдущий код в виде отдельного приложения, отметим, что выделенная очередь внутри брокера сообщений начинает получать сообщения. С другой стороны, как помним из главы 7 «Реактивный доступ к базам данных», в нашем приложении чата есть два основных потребителя сообщений: контроллер и служба статистики. На следующем шаге в реализации миграции системы изменим службу статистики. В нашем приложении служба статистики не простой потребитель, она отвечает за обновление статистики на основе изменений в базе данных и отправляет ее контроллеру. То есть эта служба является процессором `Processor`, так как играет роль источника и приемника одновременно. Следовательно, мы должны дать ей возможность получать сообщения от брокера, а также отправлять их брокеру. Рассмотрим следующий код.

```
@SpringBootApplication                             // (1)
@EnableBinding(Processor.class)                     //
/* @Service */                                      //
public class DefaultStatisticService                //
```

```

implements StatisticService {                                     //
...                                                             // (2)

@StreamListener                                                  // (3)
@Output(Processor.OUTPUT)                                       //
public Flux<UsersStatisticVM> updateStatistic(                   //
    @Input(Processor.INPUT) Flux<MessageResponse> messagesFlux // (3.1)
) { ... }                                                       //

...                                                             // (2)

public static void main(String[] args) {                        // (4)
    SpringApplication.run(DefaultStatisticService.class, args); //
}                                                                //
}

```

Пояснения к коду.

1. Объявление `@SpringBootApplication`. Здесь, как и в предыдущем примере, мы заменили `@Service` аннотацией `@EnableBinding`. В отличие от компонента `GitterService`, используем интерфейс `Processor`, объявив, что компонент `StatisticService` потребляет данные из брокера сообщений, а также отправляет их брокеру сообщений.
2. Эта часть кода не изменилась.
3. Объявление метода процессора. Здесь метод `updateStatistic` принимает поток `Flux`, который обеспечивает доступ к входящему сообщению из канала брокера сообщений. Мы должны явно определить, что данный метод принимает сообщения из брокера, добавив аннотацию `@StreamListener` вместе с аннотацией `@Input`.
4. Объявление метода `main`, который осуществляет запуск приложения `Spring Boot`.

Как видите, мы используем аннотации `Spring Cloud Streams`, только чтобы отметить, что входной и выходной потоки `Flux` являются потоками из/в определенной очереди. В этом примере `@StreamListener` позволяет имени виртуальной очереди (откуда/куда сообщения принимаются/отправляются) соответствовать имени в аннотациях `@Input/@Output`, в то время как предварительно настроенные интерфейсы связаны в аннотации `@EnableBinding`. Как показано в предыдущем примере, наряду с производителями мы можем настроить объявленные входы и выходы одним и тем же универсальным способом, используя те же свойства приложения (в данном случае конфигурацию `YAML`).

```

spring.cloud.stream:                                           //
bindings:                                                       //
  input:                                                         // (1)
    destination: Messages                                       //

```

```

        group: statistic // (2)
    output: //
    producer: //
        requiredGroups: ui //
    destination: Statistic //

```

Spring Cloud Streams обеспечивает гибкость в настройке связи с брокером сообщений. Здесь в строке (1) определяем ввод `input`, который, по сути, описывает конфигурацию потребителя. Дополнительно (2) мы должны определить имя группы `group`, представляющее имя группы получателей в брокере сообщений.



Узнать больше о доступных параметрах настройки для Spring Cloud Stream можно по ссылке https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_configuration_options.

Наконец, после подготовки эмиттеров мы должны внести изменения в компонент `InfoResource`.

```

@RestController // (1)
@RequestMapping("/api/v1/info") //
@EnableBinding({MessageSource.class, StatisticSource.class}) // (1.1)
@SpringBootApplication //
public class InfoResource { //

    ... // (2)
    /* public InfoResource( // (3)
        ChatService<MessageResponse> chatService, //
        StatisticService statisticService //
    ) { */ // (3.1)
    @StreamListener //
    public void listen( //
        @Input(MessageSource.INPUT) Flux<MessageResponse> messages, //
        @Input(StatisticSource.INPUT) //
        Flux<UsersStatisticVM> statistic //
    ) { //

    /* Flux.mergeSequential( // (4)
        chatService.getMessagesAfter("") //
        .flatMapIterable(Function.identity()), //
        chatService.getMessagesStream() //
    ) //
    .publish(flux -> Flux.merge( ... */ //

        messages.map(MessageMapper::toViewModelUnit) // (5)
        .subscribeWith(messagesStream); //
        statistic.subscribeWith(statisticStream); //
    /* }) // (4)
    .subscribe(); */ //

```



```

    }
    ...
    public static void main(String[] args) {
        SpringApplication.run(InfoResource.class, args);
    }
}

```

Пояснения к коду.

1. Определение `@SpringBootApplication`. Как можно заметить, `@EnableBinding` принимает два настраиваемых интерфейса привязки с отдельными настройками входного канала для статистики и сообщений.
2. Эта часть кода не изменилась.
3. Объявление метода `.listen`. Как видите, конструктор, который принимал два интерфейса, теперь принимает потоки с аннотацией `@Input`.
4. Изменившаяся логика. Нам больше не нужно вручную объединять потоки, потому что ответственность за это теперь несет брокер сообщений.
5. Здесь мы подписываемся на потоки статистики и сообщений. В настоящий момент все входящие сообщения кешируются в `ReplayProcessor`. Обратите внимание, что кеш является локальным, но для большей масштабируемости вместо него можно использовать распределенный кеш.
6. Объявление метода `main`, который осуществляет запуск приложения Spring Boot.



Узнать больше об интеграции с распределенными кешами, такими как Hazelcast, можно в описании модуля Reactor-Addons: <https://github.com/reactor/reactor-addons/tree/master/reactor-extra/src/main/java/reactor/cache>.

Здесь мы извлекаем сообщения из двух отдельных очередей. Использование брокера сообщений обеспечило прозрачное отделение как от `GitterService`, так и от `StatisticService`. Кроме того, имея дело с Spring Cloud Stream, мы должны помнить, что аннотация `@StreamListener` применяется только к методам. Поэтому нам пришлось пойти на хитрость, применив `@StreamListener` к методу `void`, который вызывается после установки соединения с брокером сообщений.

Чтобы лучше понять особенности связующих (bindable) интерфейсов, рассмотрим следующий код.

```

interface MessagesSource {
    String INPUT = "messages";

    @Input(INPUT)
    SubscribableChannel input();
}

```



```
} //
interface StatisticSource { //
    String INPUT = "statistic"; // (1)
    //
    @Input(INPUT) // (2)
    SubscribableChannel input(); // (3)
} //
```

Пояснения к коду.

1. Строковая константа, представляющая имя связанного канала.
2. Аннотация `@Input` объявляет, что аннотированный метод возвращает `MessageChannel`, через который входящие сообщения поступают в приложение.
3. Этот метод определяет тип `MessageChannel`. В случае связующего интерфейса потребителя сообщений мы должны определить `SubscribableChannel`, расширяющий `MessageChannel` двумя дополнительными методами для асинхронного приема сообщений.

Так же как в предыдущих случаях, мы должны определить аналогичные свойства в локальном `application.yaml`.

```
spring.cloud.stream:
  bindings:
    statistic:
      destination: Statistic
      group: ui
  messages:
    destination: Messages
    group: ui
```

Сложив все части мозаики, получаем систему с архитектурой, изображенной на рис. 8.9.

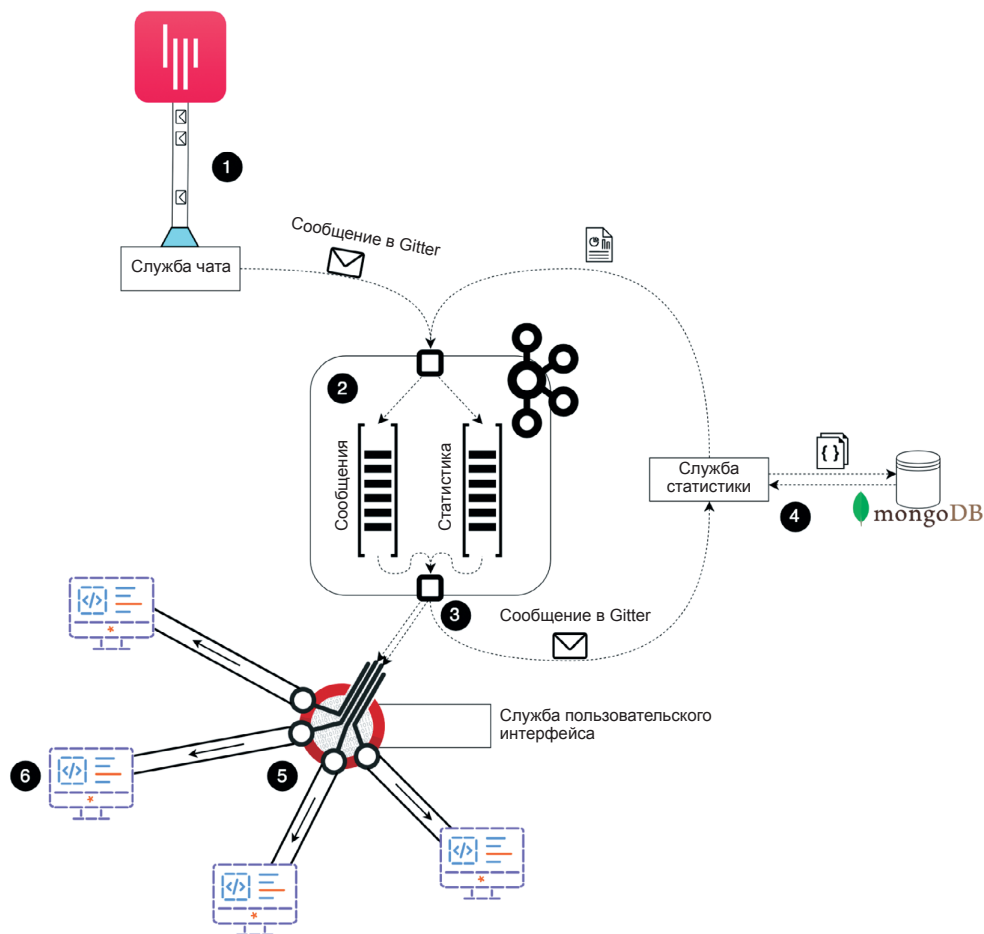


Рис. 8.9. Пример распределенного приложения чата

Рассмотрим нумерованные элементы на рис. 8.9.

1. Представление GitterService. Как можно заметить, GitterService тесно связана с Gitter API и брокером сообщений (2) (в данном примере Apache Kafka). Однако здесь нет прямой зависимости от внешних служб.
2. Представление брокера сообщений. Тут мы имеем две виртуальные очереди. Обратите внимание, что это представление не раскрывает конкретных настроек, таких как репликация или сегментирование.
3. В этой точке брокер сообщений демультиплексирует сообщение для службы пользовательского интерфейса (InfoResource) и StatisticService.
4. Представление StatisticService. Как можно заметить, служба извлекает сообщения, поступающие в брокер, сохраняет их в MongoDB, выполняет статистические вычисления и посылает обновленные результаты.

5. Обе очереди потребляются службой пользовательского интерфейса, которая рассылает сообщения всем подписавшимся клиентам.
6. Представление веб-браузера. В нашем случае клиентами службы пользовательского интерфейса являются веб-браузеры, которые получают всю информацию через HTTP-соединение.

Как можно заметить в диаграмме на рис. 8.9, наше приложение полностью разделено на уровне компонентов. Например, `GitterService`, `StatisticService` и служба пользовательского интерфейса запускаются как отдельные приложения (которые можно запустить на отдельных компьютерах) и отправляют сообщения брокеру сообщений. Кроме того, модуль Spring Cloud Streams поддерживает модель программирования Project Reactor, соответствующую стандарту Reactive Streams, и обеспечивает управление обратным давлением, а значит, более высокую надежность. При такой организации каждая служба может масштабироваться независимо, поэтому мы можем создать реактивную систему, которая является основной целью перехода на Spring Cloud Streams.

Реактивное программирование в облаке

Несмотря на то что Spring Cloud Streams предоставляют упрощенный способ создания распределенной реактивной системы, нам все равно приходится иметь дело со списком конфигураций (например, конфигураций получателей), чтобы удовлетворить все особенности модели программирования Spring Cloud Streams и т. д. Другая существенная проблема – возможность рассуждений о работе конвейера. Как рассказывалось в главе 2 «*Реактивное программирование в Spring. Основные понятия*», одной из основных причин появления реактивных расширений (как идеи асинхронного программирования) была необходимость создания инструмента, скрывающего сложный асинхронный поток данных за функциональной цепочкой операторов. Несмотря на то что мы можем разрабатывать конкретные компоненты и определять порядок взаимодействий между ними, полная картина всех взаимодействий может оставаться загадкой. Точно так же в реактивных системах понимание взаимодействий микросервисов является важнейшим и труднодостижимым без специальных инструментов аспектом.

К счастью, в 2014 году в Amazon была разработана AWS Lambda. Это добавило новые возможности в разработку реактивных систем. Об этом говорится на официальной веб-странице данной службы:

«AWS Lambda – это сервис бессерверных вычислений (<https://aws.amazon.com/serverless>), который запускает программный код в ответ на определенные события и отвечает за автоматическое выделение необходимых вычислительных ресурсов (<https://aws.amazon.com/lambda/features>)».

AWS Lambda позволяет создавать небольшие, независимые и масштабируемые преобразования. Помогает отделить жизненный цикл разработки бизнес-логики от конкретного потока данных. Наконец, удобный интерфейс дает возможность использовать каждую функцию для построения всего бизнес-процесса независимо.

Компания Amazon является пионером в этой области и вдохновила многих облачных провайдеров на использование той же технологии. К счастью, Pivotal была одной из первых, кто принял эту технологию на вооружение.

Spring Cloud Data Flow

В начале 2016 года команда Spring Cloud представила новый модуль Spring Cloud Data Flow, официально заявив в описании, которое можно найти по ссылке <https://cloud.spring.io/spring-cloud-dataflow>:

«Spring Cloud Data Flow предлагает набор инструментов для интеграции данных и конвейеров их обработки в масштабе реального времени».

Говоря отвлеченно, основная идея этого модуля заключается в достижении разделения между разработкой функциональных преобразований и фактическим взаимодействием между разработанными компонентами. Другими словами, это разделение функций и их сочетаний в бизнес-процессе. Для решения данной проблемы Spring Cloud Data Flow предлагает удобный веб-интерфейс, который позволяет загружать разворачиваемые приложения Spring Boot. Затем он настраивает потоки данных, используя загруженные артефакты и разворачивая сконструированный канал на выбранную платформу, такую как Kubernetes, Apache YARN или Mesos. Кроме того, Spring Cloud Data Flow предлагает широкий выбор готовых коннекторов для разнообразных источников данных (баз данных, очередей сообщений и файлов), различных встроенных процессоров для преобразования данных и приемников, обеспечивающих различные способы хранения результатов.



Узнать больше о поддерживаемых источниках данных, процессорах и приемниках можно по ссылкам <https://cloud.spring.io/spring-cloud-task-app-starters/> и <https://cloud.spring.io/spring-cloud-stream-app-starters/>.

Как уже отмечалось, Spring Cloud Data Flow использует идею потоковой обработки. Следовательно, все разворачиваемые конвейеры конструируются на основе модуля Spring Cloud Stream, а все взаимодействия осуществляются через распределенные брокеры сообщений, такие как Kafka, или распределенные и высокомасштабируемые варианты RabbitMQ.

Чтобы понять возможности распределенного реактивного программирования с использованием Spring Cloud Data Flow, создадим конвейер обработки платежей. Как вы уже знаете, обработка платежей – весьма сложный процесс, мы же сосредоточимся на его упрощенной версии, изображенной на рис. 8.10.

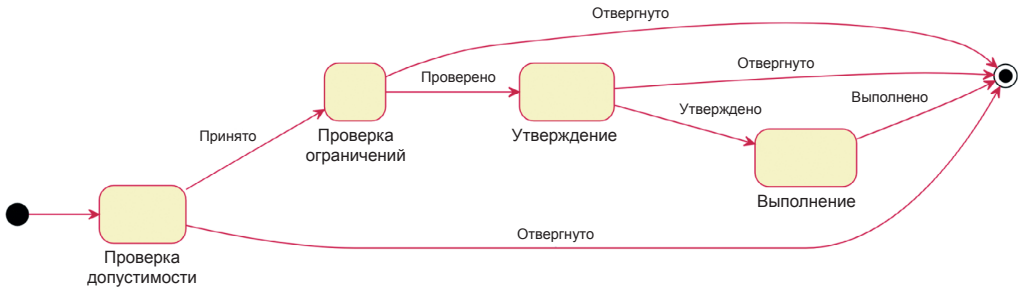


Рис. 8.10. Конвейер обработки платежей

Платеж пользователя должен пройти через несколько важных этапов, таких как проверка допустимости, проверка ограничений и утверждение. В главе 6 «*Неблокирующие и асинхронные взаимодействия с WebFlux*» мы создали аналогичное приложение, где весь процесс организовывала единственная служба. Несмотря на то что все взаимодействия осуществлялись с использованием асинхронных вызовов нескольких независимых микросервисов, состояние потока поддерживалось одной службой, реализованной на основе Reactor 3. Это означает, что в случае сбоя службы восстановление состояния может оказаться сложной задачей.

К счастью, Spring Cloud Data Flow опирается на Spring Cloud Streams, который, в свою очередь, основывается на отказоустойчивом брокере сообщений. Следовательно, в случае сбоя брокер сообщений не подтвердит получение сообщения, и оно будет доставлено другому исполнителю без дополнительных усилий.

Поскольку мы уже имеем базовое понимание внутреннего устройства Spring Cloud Data Flow и основных бизнес-требований к конвейеру обработки платежей, можем реализовать эту службу, используя данный стек технологий.

Прежде всего мы должны определить точку входа, которая обычно доступна как конечная точка HTTP. Spring Cloud Data Flow предлагает источник HTTP, который можно определить на предметном языке **Spring Cloud Data Flow DSL**:

```
SendPaymentEndpoint=Endpoint: http --path-pattern=/payment --port=8080
```



В этом примере представлена малая часть каналов Spring Cloud Data Flow. Далее вы увидите больше примеров, демонстрирующих конструирование полных каналов Spring Cloud Data Flow. Узнать больше о Stream Pipeline DSL можно по ссылке <https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#spring-cloud-dataflow-stream-intro-dsl>.

Перед началом любых манипуляций убедитесь, что все поддерживаемые приложения и задачи уже зарегистрированы здесь: <https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#supported-apps-and-tasks>.

В предыдущем примере мы определили новую функцию конвейера обработки данных, которая представляет все HTTP-запросы в виде потока сообщений. Следовательно, можем реагировать на них определенным образом.

Модульная организация приложений с Spring Cloud Function

После определения конечной точки HTTP мы должны проверить входящее сообщение. К сожалению, эта часть процесса содержит актуальную бизнес-логику и требует конкретной реализации. К счастью, Spring Cloud Data Flow позволяет использовать пользовательские приложения Spring Cloud Stream как части процесса.



Мы не будем вдаваться в детали реализации бизнес-этапов в Spring Cloud Data Flow. Узнать больше о создании и регистрации своих приложений Spring Boot можно по следующим ссылкам:

- <https://docs.spring.io/spring-cloud-dataflow/payment/docs/current/reference/htmlsingle/#custom-applications>;
- https://docs.spring.io/spring-cloud-dataflow-samples/docs/current/reference/htmlsingle/#_custom_spring_cloud_stream_processor.

С одной стороны, мы можем предоставить наше собственное отдельное приложение Spring Cloud Stream с настраиваемой логикой проверки. С другой стороны, нам все еще приходится иметь дело со всеми конфигурациями, uber-jar-файлами, длительным временем запуска и остальными проблемами, связанными с развертыванием приложения. К счастью, таких проблем можно избежать с помощью проекта Spring Cloud Function.

Основная цель Spring Cloud Function – продвижение бизнес-логики с помощью функций. Этот проект предлагает возможность отделить пользовательскую бизнес-логику от особенностей времени выполнения. Следовательно, одну и ту же функцию можно повторно использовать разными способами и в разных местах.



Узнать больше о возможностях проекта Spring Cloud Function можно по ссылке <https://github.com/spring-cloud/spring-cloud-function>.

Прежде чем начать работать с Spring Cloud Function, рассмотрим основные возможности модуля Spring Cloud Function и разберемся с его внутренним устройством.

Фактически Spring Cloud Function – это дополнительный уровень абстракции для приложений, которые могут выполняться поверх Spring Cloud Streams, AWS Lambda или любой другой облачной платформы, используя любой транспорт для передачи данных.

По умолчанию Spring Cloud Function имеет адаптеры для настройки развертывания функций в AWS Lambda, Azure Functions и Apache OpenWhisk. Основным преимуществом использования Spring Cloud Function перед прямой загрузкой Java-функций является возможность использовать большинство возможностей Spring и не зависеть от конкретного SDK поставщика облачных услуг.

Модель программирования, предлагаемая Spring Cloud Function, является не более чем определением одного из следующих классов Java – `java.util.function.Function`, `java.util.function.Supplier` и `java.util.function.Consumer`. Кроме того, Spring Cloud Function можно использовать в различных комбинациях фреймворков. Например, можно создать приложение Spring Boot, служащее платформой для элемента функций. Некоторые из них, в свою очередь, могут быть представлены как обычный Spring-компонент `@Bean`.

```
@SpringBootApplication                                // (1)
@EnableBinding(Processor.class)                        // (1.1)
public class Application {                             //
    @Bean                                              // (2)
    public Function<
        Flux<Payment>,
        Flux<Payment>
    > validate() {                                     //
        return flux -> flux.map(value -> { ... });    // (2.1)
    }                                                 //

    public static void main(String[] args) {          // (3)
        SpringApplication.run(Application.class, args);
    }                                                 //
}
```

Пояснения к коду.

1. Объявление `@SpringBootApplication`. Как можно заметить, мы все еще должны использовать минимальный набор объявлений для приложения Spring Boot. Также мы используем аннотацию `@EnableBinding` с интерфейсом `Processor` в качестве параметра. В такой комбинации Spring Cloud идентифицирует компонент в строке (2) как обработчик сообщений. Кроме того, *вход* и *выход* функции связаны с внешними адресатами, сопряженными с процессором `Processor`.
2. Объявление функции `Function`, которая преобразует один поток данных `Flux` в другой, как компонент контейнера IoC. В строке (2.1) объявляем лямбду для проверки элементов, которая, в свою очередь, является функцией высшего порядка, принимающей один поток данных и возвращающей другой.
3. Объявление метода `main`, который осуществляет запуск приложения Spring Boot.

Как можно заметить в предыдущем примере, Spring Cloud Function поддерживает разные модели программирования, например преобразование сообщений с поддержкой реактивных типов Reactor 3 и Spring Cloud Streams, которое открывает доступ к этим потокам внешним адресатам.

Кроме того, Spring Cloud Function не ограничивается предопределенными функциями. Например, он имеет встроенный компилятор времени выполнения, который позволяет передать функцию в виде строки в файле свойств, как показано в следующем примере:

```
spring.cloud.function:                // (1)
  compile:                            // (2)
    payments:                          // (3)
      type: supplier                   // (4)
      lambda: ()->Flux.just(new org.TestPayment()) // (5)
```

Пояснения к коду.

1. Пространство имен свойств Spring Cloud Function.
2. Пространство имен, связанное с компиляцией функции во время выполнения.
3. Определение ключа – имени, под которым функция будет видна в контейнере Spring IoC. Также играет роль имени файла для скомпилированного байт-кода.
4. Определение типа функции. Возможные варианты: `supplier/function/consumer`.
5. Определение лямбды. Как видите, производитель (`supplier`) определяется как строка, которая компилируется в байт-код и сохраняется в файловой системе. Компиляция осуществляется модулем `spring-cloud-function-compiler`. Он включает встроенный компилятор и позволяет сохранять скомпилированные функции и добавлять их в `ClassLoad`.

Предыдущий пример показывает, что Spring Cloud Function предлагает возможность динамически определять и запускать функции без их предварительной компиляции. Эту возможность можно использовать для реализации **функции как услуги** (Function as a Service, FaaS) в программном решении.

Наряду с этим Spring Cloud Function предлагает модуль `spring-cloud-function-task`, помогающий запускать упомянутые функции в конвейере с использованием того же файла свойств.

```
spring.cloud.function:
  task:                                // (1)
    supplier: payments                 // (2)
    function: validate|process         // (3)
    consumer: print                    // (4)
```



```

compile:
  print:
    type: consumer
    lambda: System.out::println
    inputType: Object
process:
  type: function
  lambda: (flux)->flux
  inputType: Flux<org.rpis5.chapters.chapter_08.scf.Payment>
  outputType: Flux<org.rpis5.chapters.chapter_08.scf.Payment>

```

Пояснения к коду.

1. Пространство имен, используемое для настроек задач.
2. Настройки функции `supplier` (производителя) для задачи. Чтобы определить производителя, нужно передать имя соответствующей функции.
3. Промежуточные преобразования данных. Здесь можно объединить в конвейер несколько функций, используя символ вертикальной черты (`|`). Обратите внимание, что функции объединяются в цепочку с применением метода `Function#accept`.
4. Определение этапа потребления. Задача может быть выполнена только после определения всех этапов.

Как видим, в приложении Spring Boot с подключенными модулями Spring Cloud Function можно выполнять функции, подготовленные пользователями, и объединять их в сложные цепочки обработки.

Важную роль в экосистеме Spring Cloud Function играет модуль `spring-cloud-function-compiler`. Наряду с компиляцией функций из файла свойств этот модуль предлагает конечные точки, которые позволяют разворачивать функции на лету. Например, вызвав следующую команду `curl` в терминале, можно добавить указанную функцию в работающее приложение Spring Boot:

```

curl -X POST -H "Content-Type: text/plain" \
  -d "f->f.map(s->s.toUpperCase())" \
  localhost:8080/function/uppercase\
  ?inputType=Flux%3CString%3E\
  &outputType=Flux%3CString%3E

```

В этом примере мы выгрузили функцию, принимающую и возвращающую `Flux<String>`.



Обратите внимание, что здесь последовательности символов `%3C` и `%3E` кодируют угловые скобки `<` и `>`, как принято в адресах HTTP URL.



Есть два способа запустить `spring-cloud-function-compiler` в роли сервера:

- загрузить JAR-файл из репозитория `maven-central` и запустить его независимо;
- добавить модуль как зависимость в проект, указав следующий путь для поиска компонентов: `"org.springframework.cloud.function.compiler.app"`.

После развертывания функции путем запуска облегченного приложения Spring Boot с зависимостями `spring-cloud-function-web` и `spring-cloud-function-compiler` можем развернуть функцию через HTTP на лету и в виде отдельного веб-приложения. Например, один и тот же файл `jar`, меняя аргументы программы, можно запустить с разными функциями, как показано ниже:

```
java -jar function-web-template.jar \  
  --spring.cloud.function.imports.uppercase.type=function \  
  --spring.cloud.function.imports.uppercase.location=\   
  file:///tmp/function-registry/functions/uppercase.fun \  
 \  
  --spring.cloud.function.imports.worldadder.type=function \  
  --spring.cloud.function.imports.worldadder.location=\   
  file:///tmp/function-registry/functions/worldadder.fun
```

В этом примере импортируем две функции:

- **uppercase**, преобразующую все символы в строке в верхний регистр;
- **worldadder**, добавляющую окончание `world` к заданной строке.

Как можно заметить, здесь используем пространство имен `spring.cloud.function.imports` для определения имен импортируемых функций (выделены **жирным** шрифтом), их типов (выделены *курсивом*) и местоположения байт-кода этих функций. После успешного запуска приложения получим доступ к развернутым функциям, выполнив следующую команду `curl`:

```
curl -X POST -H "Content-Type: text/plain" \  
  -d "Hello" \  
  localhost:8080/uppercase%7Cworldadder
```

В результате получаем строку `"HELLO World"`, которая наглядно доказывает, что обе функции присутствуют на сервере и выполнены в порядке их следования в URL.



Символ `%7C` здесь представляет символ вертикальной черты (`|`), как принято в адресах HTTP URL.

Аналогично можно развертывать и импортировать другие функции, находящиеся в том же или в другом приложении.

Кроме того, Spring Cloud Function предлагает модуль развертывания, который играет роль контейнера для независимых функций. В предыдущих случаях мы могли запускать встроенные или развертывать свои функции через веб-API `spring-cloud-function-compiler`. Мы видели, как использовать развернутые функции и запускать их как независимые приложения. Несмотря на эту гибкость, время запуска приложения Spring Boot может оказаться намного больше времени выполнения самой функции. В некоторых случаях (наряду с чистыми функциями) мы должны использовать кое-что из арсенала Spring Framework, например возможности Spring Data или Spring Web, что может пригодиться в таких случаях, как размещение тонких jar-файлов. Spring Cloud Function также предлагает дополнительный модуль `spring-cloud-function-deployer`.

Модуль **Spring Cloud Function Deployer** позволяет запускать каждый jar-файл с одним и тем же приложением Spring Deployer, но в полной изоляции. На первый взгляд, этот модуль не дает никаких ценных выгод. Однако, как мы помним, независимые функции быстро инициализируются и выполняются. Чтобы запустить функцию, упакованную в среду Spring Boot, требуется запустить все приложение Spring Boot, что обычно занимает значительное время по сравнению со временем запуска функции.

Для решения этой проблемы Spring Cloud Function Deployer сначала запускается сам и предварительно загружает некоторую часть классов JDK. Затем создает дочерний `ClassLoader` для каждого jar-файла с функциями. Выполнение каждого jar-файла происходит в отдельном потоке, что обеспечивает параллельное выполнение. Поскольку каждый jar-файл является независимым микроприложением Spring Boot, он работает в собственном контексте Spring, поэтому их компоненты не пересекаются с компонентами соседних приложений. В результате запуск дочернего приложения Spring Boot происходит значительно быстрее, поскольку родительский `ClassLoader` уже выполнил сложную работу по разогреву JVM.

Более того, убийственная комбинация `spring-cloud-function-deployer` и `spring-boot-thin-launcher` позволяет также решить проблему **толстых jar-файлов**. **Spring Boot Thin Launcher** – это плагин для Maven и Gradle, который переопределяет стандартную версию `JarLauncher` из Spring Boot и предлагает вместо него `ThinJarWrapper` и `ThinJarLauncher`. Эти классы выполняют всю работу, необходимую для упаковки jar-файла без зависимостей, а затем – только на этапе начальной загрузки – они находят все необходимые зависимости в настроенном кеше (например, в локальном репозитории Maven) или загружают отсутствующие зависимости из настроенного репозитория Maven. Действуя подобным образом, приложение может уменьшить размер jar-файла до нескольких килобайт, а время запуска – до сотен миллисекунд.



Узнать больше о Thin Launcher и о преимуществах, предлагаемых модулем Spring Cloud Function Deployer, можно по ссылкам <https://github.com>.

`com/dsyer/spring-boot-thin-launcher`, <https://github.com/dsyer/spring-boot-thin-launcher/tree/master/deployer> и <https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-deployer>.

Завершая обсуждение Spring Cloud Function, рассмотрим обобщенную диаграмму экосистемы на рис. 8.11.

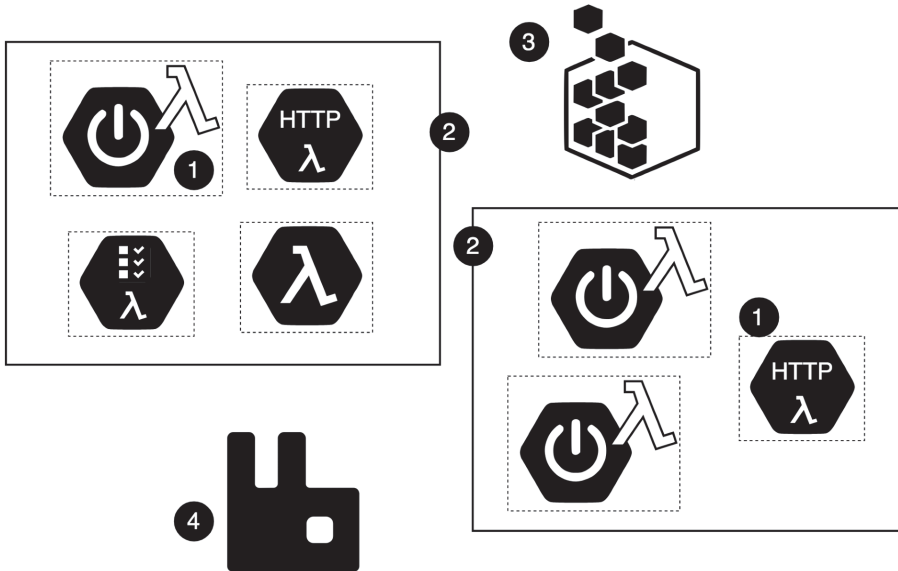


Рис. 8.11. Экосистема Spring Cloud Function

Рассмотрим нумерованные элементы на рис. 8.11.

1. Шестиугольники – это функции. Как видите, здесь изображено несколько разных типов шестиугольников. Некоторые из них представляют комбинации функций в приложении Spring Boot или функций, доступных через HTTP. Другие приложения могут взаимодействовать с функциями посредством Spring Cloud Stream Adapter или развертывать их как задачи для однократного выполнения.
2. Модуль Spring Cloud Function Deployer. Как упоминалось выше, он изображается в виде контейнера. В этом случае у нас есть два независимых контейнера Spring Cloud Function Deployer, выполняемых на разных узлах. Пунктирные границы вокруг функций внутри контейнеров представляют независимые экземпляры ClassLoader.
3. Модуль Spring Cloud Function Compiler. В данном случае модуль играет роль сервера, который позволяет развертывать функции по HTTP и хранить их в хранилище.
4. Брокер сообщений, роль которого в данном случае играет RabbitMQ.

Таким образом, используя модули Spring Cloud Function и интеграцию с существующими облачными платформами, можно создать свою платформу **функция как служба** (Function as a Service, FaaS), которая предлагает почти весь арсенал возможностей Spring Framework, что позволяет создавать приложения с использованием легковесных функций. Однако мы должны помнить, что Spring Cloud Function демонстрирует силу, когда имеется основа для развертывания, мониторинга и управления экземплярами, поэтому экосистему Spring Cloud Function можно построить поверх данной основы. С этой целью в следующем разделе рассмотрим, как работает Spring Cloud Function в сочетании с полноценной экосистемой Spring Cloud Data Flow.

Spring Cloud – функция как часть конвейера обработки данных

Теперь, имея достаточно знаний об экосистеме Spring Cloud Function, можем вернуться к нашей теме и посмотреть, как использовать этот замечательный модуль. Существует дополнительный модуль под названием **Spring Cloud Starter Stream App Function**, который позволяет использовать возможности Spring Cloud Function в Spring Cloud Data Flow. Он позволяет использовать чистые файлы jar и развертывать их как часть Spring Cloud Data Flow без избыточных издержек Spring Boot. Поскольку у нас есть прямое соответствие, упрощенная функция Validation может быть сведена к обычной функции `Function<Payment, PaymentValidation>`, как показано ниже.

```
public class PaymentValidator
    implements Function<Payment, Payment> {

    public Payment apply(Payment payment) { ... }
}
```

После упаковки и публикации артефакта можем написать следующий сценарий потокового канала, соединяющего наш HTTP-источник с мостом Spring Cloud Function Bridge.

```
SendPaymentEndpoint=Endpoint: http --path-pattern=/payment
--port=8080 | Validator: function --class-name=com.example.
PaymentValidator --location=https://github.com/PacktPublishing/Hands-
On-Reactive-Programming-in-Spring-5/tree/master/chapter-08/dataflow/
payment/src/main/resources/payments-validation.jar?raw=true
```



На момент написания этих строк модуль Spring Cloud Function для Spring Cloud Data Flow не подключался по умолчанию в пакеты Applications и Tasks, и его необходимо было регистрировать с помощью следующих свойств импорта:

```
source.function=maven://org.springframework.cloud.stream.  
app:function-app-rabbit:1.0.0.BUILD-SNAPSHOT  
  
source.function.metadata=maven://org.springframework.cloud.stream.  
app:function-app-rabbit:jar:metadata:1.0.0.BUILD-SNAPSHOT  
  
processor.function=maven://org.springframework.cloud.stream.  
app:function-app-rabbit:1.0.0.BUILD-SNAPSHOT  
  
processor.function.metadata=maven://org.springframework.cloud.  
stream.app:function-app-rabbit:jar:metadata:1.0.0.BUILD-SNAPSHOT  
  
sink.function=maven://org.springframework.cloud.stream.  
app:function-app-rabbit:1.0.0.BUILD-SNAPSHOT  
  
sink.function.metadata=maven://org.springframework.cloud.stream.  
app:function-app-rabbit:jar:metadata:1.0.0.BUILD-SNAPSHOT
```

Наконец, чтобы завершить первую часть процесса, после проверки допустимости мы должны доставить платеж в разные пункты, выбрав конечную точку в зависимости от результата проверки. Поскольку функция проверки является чистой функцией, которая не должна иметь доступа к инфраструктуре (такой как заголовки маршрутизации RabbitMQ), мы должны делегировать эту ответственность кому-то еще. К счастью, Spring Cloud Data Flow предлагает функцию Router Sink, которая позволяет пересылать входящие сообщения в разные очереди, опираясь на следующее выражение:

```
... | router --expression="payload.isValid() ? 'Accepted' : 'Rejected'"
```

Также можем настроить источник, принимающий сообщения из очереди с определенным именем. Например, вот как мог бы выглядеть сценарий, отвечающий за получение сообщений из канала RabbitMQ с именем Accepted:

```
...  
Accepted=Accepted: rabbit --queues=Accepted
```

Согласно диаграмме потока обработки платежей, следующий шаг – сохранение платежа со статусом «Принят» (Accepted). То есть пользователи должны иметь возможность посетить определенную страницу со своими платежами и проверить состояние обработки каждого платежа. А значит, мы должны обеспечить интеграцию с базой данных. Например, можем хранить переходы платежа из состояния в состояние в MongoDB. Spring Cloud Data Flow предлагает приемник *MongoDB Sink*. Используя его, можно легко записывать входящие сообщения в MongoDB. Добавив Spring Data Flow, можем передавать сообщения как в приемник MongoDB Sink, так и на следующий этап выполнения. Эту методику можно использовать только при работе с абсолютно надежным брокером сообщений, таким как Apache Kafka. Как известно, Kafka сохраняет сообщения в хранилище. Следовательно, сообщения будут доступны в брокере сообщений, даже если на каком-то этапе произойдет сбой. Таким образом, MongoDB хранит состояние, пред-

назначенное для отображения в пользовательском интерфейсе, тогда как фактическое состояние обработки хранится внутри брокера сообщений. Следовательно, оно может быть восстановлено в любой момент времени. С другой стороны, в случае быстрых брокеров сообщений в памяти, таких как RabbitMQ, было бы достаточно положиться на состояние, хранящееся в MongoDB, в качестве источника истины. А значит, мы должны убедиться, что состояние платежа сохранено до перехода к следующему этапу. К сожалению, для этого нам нужно написать собственное приложение Spring Cloud Stream, обертывающее MongoDB как один из этапов обработки.

Повторив аналогичные операции для остальной части процесса, можно получить процесс, изображенный на рис. 8.12.

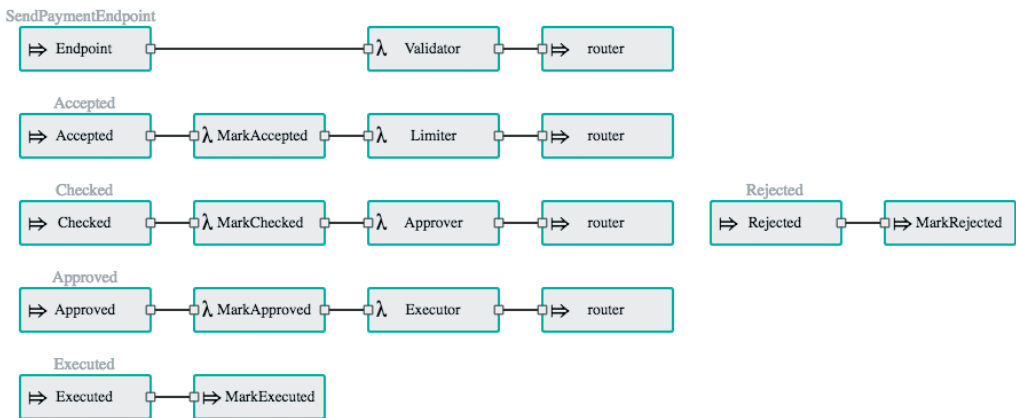


Рис. 8.12. Полный процесс обработки платежа с поддержкой пользовательского интерфейса



На рис. 8.12 изображен встроенный дашборд Spring Cloud Data Flow, позволяющий конструировать процессы и управляющие приложения с использованием веб-интерфейса. Не будем углубляться в подробное описание этого дашборда, но вы можете узнать больше, пройдя по ссылкам <https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#dashboard>, <https://github.com/spring-projects/spring-flo/> и <https://github.com/spring-cloud/spring-cloud-dataflow-ui>. Кроме дашборда имеется также клиент командной строки, предлагающий те же возможности. Узнать больше об этом клиенте можно по ссылке <https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#shell>.

Изображение для рис. 8.12 получено с помощью следующего сценария:

```

SendPaymentEndpoint=Endpoint: http --path-pattern=/payment
--port=8080 | Validator: function --class-name=com.example.
PaymentValidator --location=https://github.com/PacktPublishing/

```



```
Hands-On-Reactive-Programming-in-Spring-5/tree/master/chapter-08/
dataflow/payment/src/main/resources/payments.jar?raw=true | router
--expression="payload.isValid() ? 'Accepted' : 'Rejected'"
```

```
Accepted=Accepted: rabbit --queues=Accepted | MarkAccepted: mongodb-
processor --collection=payment | Limiter: function --class-name=com.
example.PaymentLimiter --location=https://github.com/PacktPublishing/
Hands-On-Reactive-Programming-in-Spring-5/tree/master/chapter-08/
dataflow/payment/src/main/resources/payments.jar?raw=true | router
--expression="payload.isLimitBreached() ? 'Rejected' : 'Checked'"
```

```
Checked=Checked: rabbit --queues=Checked | MarkChecked: mongodb-
processor --collection=payment | Approver: function --class-name=com.
example.PaymentApprover --location=https://github.com/PacktPublishing/
Hands-On-Reactive-Programming-in-Spring-5/tree/master/chapter-08/
dataflow/payment/src/main/resources/payments.jar?raw=true | router
--expression="payload.isApproved() ? 'Approved' : 'Rejected'"
```

```
Approved=Approved: rabbit --queues=Approved | MarkApproved: mongodb-
processor --collection=payment | Executor: function --class-name=com.
example.PaymentExecutor --location=https://github.com/PacktPublishing/
Hands-On-Reactive-Programming-in-Spring-5/tree/master/chapter-08/
dataflow/payment/src/main/resources/payments.jar?raw=true | router
--expression="payload.isExecuted() ? 'Executed' : 'Rejected'"
```

```
Executed=Executed: rabbit --queues=Executed | MarkExecuted: mongodb
--collection=payment
```

```
Rejected=Rejected: rabbit --queues=Rejected | MarkRejected: mongodb
--collection=payment
```

Наконец, развернув этот процесс, мы можем выполнить платеж и исследовать сообщения, выводимые в консоль.



Чтобы выполнить этот код в установленном сервере Spring Cloud Data Flow, у вас также должны быть установлены RabbitMQ и MongoDB.

Особенно примечательно, что процесс развертывания так же прост, как разработка бизнес-логики. Прежде всего инструменты Spring Cloud Data Flow основываются на модуле Spring Cloud Deployer, который используется для развертывания на современных платформах, таких как Cloud Foundry, Kubernetes, Apache Mesos или Apache YARN. Инструментарий предоставляет API-интерфейсы Java, которые позволяют настраивать источник приложения (например, репозиторий Maven, artifactId, groupId и version) и его последующее развертывание на целевой платформе. Наряду с этим Spring Cloud Deployer достаточно гибок и предлагает широкий список конфигураций и свойств, в том числе количество реплик развертываемого экземпляра.



Высокая доступность, отказоустойчивость или надежность развернутой группы экземпляров приложения напрямую зависят от платформы и самого Spring Cloud Deployer, который не дает никаких гарантий. Например, не рекомендуется использовать *Spring Cloud Deployer Local* в промышленных системах. Локальная версия инструментария предназначена для запуска на одной машине с использованием Docker. Отметим, что Spring Cloud Deployer SPI не имеет дополнительных средств мониторинга или обслуживания и предполагает, что все необходимые функции реализует базовая платформа.

Используя упомянутые возможности, Spring Cloud Data Flow обеспечивает развертывание одним щелчком (или одной командой в терминале) с возможностью передачи требуемых конфигураций и свойств.

Итак, мы начали с изучения основ Spring Cloud Streams и закончили рассказом о мощной абстракции в лице нескольких модулей. Мы увидели, что при поддержке этих проектов можно построить реактивную систему, применяя различные абстракции реактивного программирования. Упомянутый метод использования брокера сообщений для асинхронного и надежного обмена сообщениями покрывает большинство потребностей бизнеса. Более того, этот метод может снизить стоимость разработки реактивной системы в целом и использоваться для быстрой разработки систем, таких как большие интернет-магазины, IoT или приложения для чата. Однако важно помнить, что хотя описанный подход повышает надежность, масштабируемость и пропускную способность системы, он может увеличить задержку в процессе обработки запроса из-за дополнительных связей, особенно с брокерами сообщений, обеспечивающими хранение сообщений в базе данных. Итак, если система допускает возможность задержки на несколько дополнительных миллисекунд между отправкой и получением сообщений, то такой подход может использоваться для ее построения.

RSocket для реактивной передачи сообщений с низкой задержкой

В предыдущем разделе мы узнали, как легко создать реактивную систему, используя Spring Cloud Streams и его вариант в Spring Cloud Data Flow. Также увидели, как с помощью Spring Cloud Function построить модульную систему, использующую легковесные функции и объединяющую их в конвейер.

Однако одним из недостатков такой простоты и гибкости является увеличение задержек. В настоящее время есть сферы, где каждая миллисекунда на счету. Например, фондовые рынки, онлайн-видеоигры или системы управления производством в реальном времени. В таких системах недопустимо тратить время на

организацию очередей и извлечение сообщений из них. Кроме того, как показано выше, наиболее надежные брокеры сообщений сохраняют сообщения, что еще больше увеличивает время доставки сообщения.

Одним из возможных решений, помогающих уменьшить задержки во взаимодействиях между службами в распределенной системе, является использование постоянного прямого соединения между службами. Например, связывая приложения постоянными TCP-соединениями, можно добиться уменьшения задержек и обеспечить определенные гарантии доставки в зависимости от выбранного транспорта. Наряду с этим использование более широко известных протоколов, таких как WebSocket, позволяет создавать такие каналы связи, например, с помощью `ReactorNettyWebSocketClient` из `Spring WebFlux`.

Однако, как рассказывалось выше в этой главе, кроме образования тесной связи между службами использование WebSocket не соответствует требованиям реактивных систем, поскольку протокол не дает возможности управлять обратным давлением, которая является жизненно важной частью упругой системы.

К счастью, члены группы, стоящей за разработкой стандарта Reactive Streams, понимают необходимость асинхронной связи с низкой задержкой. В середине 2015 года Бен Кристенсен (Ben Christensen) с группой экспертов инициировал новый проект под названием RSocket.

Основная цель проекта RSocket – предоставить прикладной протокол с семантикой реактивных потоков через асинхронную двоичную границу.



Узнать больше о предпосылках создания проекта RSocket можно на странице <https://github.com/rssocket/rssocket/blob/master/Motivations.md>.

RSocket и Reactor-Netty

На первый взгляд RSocket не особенно инновационный. У нас уже есть веб-серверы, такие как RxNetty и Reactor-Netty (обертка WebFlux по умолчанию для Netty). Эти решения предлагают API (чтения/записи из/в Сеть), построенные на основе реактивных потоков, что подразумевает поддержку обратного давления. Однако единственная проблема с таким обратным давлением заключается в том, что оно работает изолированно. Как следствие надлежащая поддержка обратного давления имеется только между компонентом и сетью.

Используя Reactor-Netty, например, мы можем получать входящие байты, только когда готовы сделать это. Аналогично готовность сети определяется вызовом метода `Subscription#request`. Главная проблема заключается в том, что фактическая потребность компонентов не пересекает границы сети, как показано на рис. 8.13.

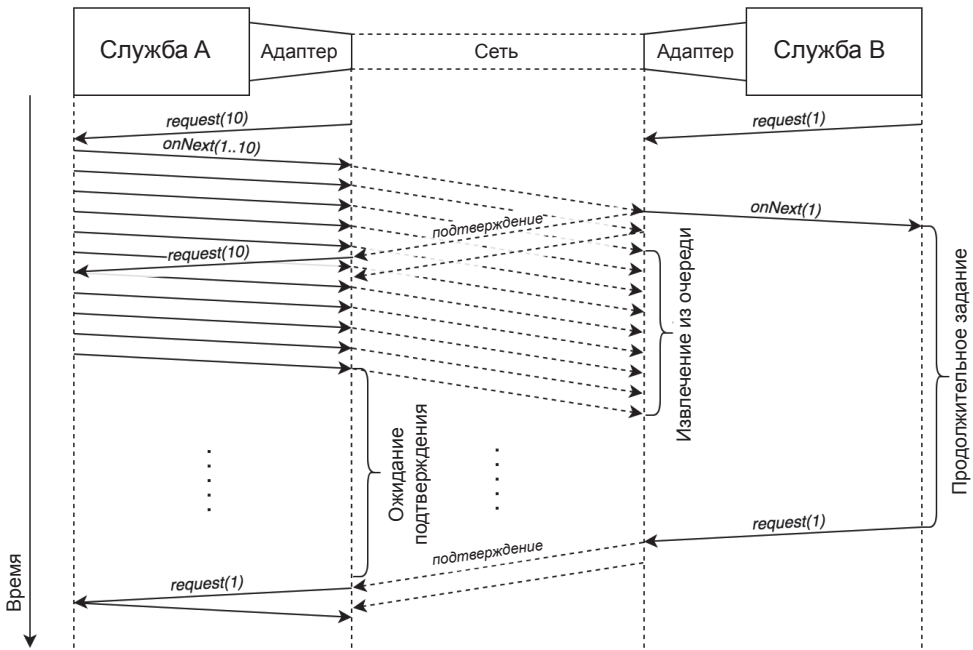


Рис. 8.13. Пример обратного давления в изоляции

Как видно на рис. 8.13, здесь есть две службы (**Служба А** и **Служба В**). Кроме того, у нас есть абстрактный **Адаптер**, который позволяет службе (бизнес-логике) взаимодействовать с другой службой по сети. Предположим, роль адаптера играет Reactor-Netty, обеспечивающий реактивный доступ. Это означает, что управление обратным давлением реализовано на транспортном уровне. Следовательно, адаптер информирует издателя, сколько элементов он может записать в поток в данный момент. В нашем случае **Служба А** создает постоянное соединение (например, соединение WebSocket) со **Службой В** и, как только соединение становится доступным, начинает отправлять элементы адаптеру. Адаптер, в свою очередь, заботится об отправке элементов в Сеть. На противоположной стороне соединения у нас есть **служба В**, которая получает сообщения из Сети через тот же адаптер. Будучи подписчиком, **служба В** передает требование в адаптер и после получения необходимого количества байтов адаптер преобразует их в логический элемент, который затем передает бизнес-логике. Аналогично после передачи элемента адаптеру тот преобразует его в байты и, следуя логике, определяемой транспортным уровнем, отправляет их в Сеть.

Как рассказывалось в главе 3 «*Reactive Streams – новый стандарт потоков*», наихудший сценарий во взаимодействиях между производителем и потребителем возникает, когда потребитель обрабатывает данные медленнее, чем производитель может их посылать, что может привести к переполнению потребителя. Что-

бы подчеркнуть проблему изолированного обратного давления, предположим, что у нас медленный потребитель и быстрый производитель, поэтому некоторая часть событий буферизируется в **Socket Buffer**. К сожалению, размер буфера ограничен, и в какой-то момент сетевые пакеты могут начать отбрасываться. Конечно, поведение транспорта во многих случаях зависит от протокола взаимодействия. Например, надежные сетевые транспорты, такие как TCP, имеют средства управления потоком (управления обратным давлением), охватывающие такие понятия, как *скользящее окно* и *подтверждение*. Это означает, что в общем случае обратное давление достижимо на двоичном уровне (на уровне принятых байтов и соответствующего подтверждения). В таком случае TCP заботится о повторной доставке сообщений и замедляет работу издателя на стороне **службы А**. Недостатком здесь является значительное влияние на производительность самих приложений, поскольку приходится выполнять повторную доставку сброшенных пакетов. Кроме того, производительность связи тоже может уменьшиться, соответственно, общая стабильность системы ухудшится. Несмотря на то что такое управление транспортным потоком в некоторых ситуациях может быть приемлемым, эффективность использования соединения оказывается слишком низкой. Именно поэтому мы не можем повторно использовать одно и то же соединение для мультиплексирования нескольких логических потоков. Другой аспект – когда потребитель не может управлять транспортным потоком и вынужден буферизовать байты у себя, что может привести к исчерпанию памяти.



Узнать больше об управлении потоками TCP можно по ссылке <https://hpbn.co/building-blocks-of-tcp/#flow-control>.

В отличие от изолированной реактивной связи между издателем и подписчиком, которая легко реализуется с помощью Reactor Netty или RxNetty, RSocket предлагает двоичный протокол, являющийся прикладным протоколом с семантикой реактивных потоков, пересекающим асинхронную границу (рис. 8.14).

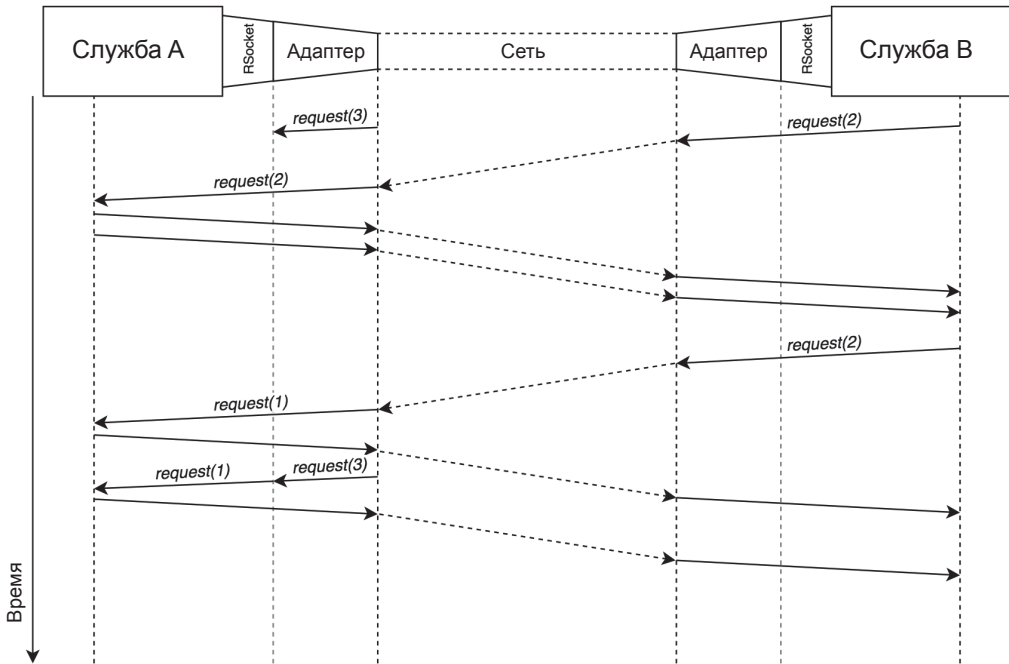


Рис. 8.14. Пример управления обратным давлением с RSocket

Как показано на рис. 8.14, протокол RSocket дает нам поддержку передачи требований через сетевые границы. Соответственно, служба-производитель может реагировать на запрос и отправлять соответствующее количество сигналов `onNext`, которые также передаются через Сеть. Кроме того, RSocket работает поверх адаптера, который заботится о записи данных в Сеть, так что локальный спрос может координироваться изолированно с запросом, полученным от службы-потребителя. В целом RSocket не зависит от выбора транспорта и наряду с TCP поддерживает Aeron и WebSocket.

На первый взгляд кажется, что использование соединения неэффективно, поскольку объем взаимодействий между службами может быть невелик. Однако одна из мощных особенностей протокола RSocket как раз заключается в том, что он позволяет повторно использовать одно и то же соединение для передачи множества потоков данных между тем же сервером и клиентом. Следовательно, есть возможность оптимизировать использование одного соединения.

Наряду с протоколом RSocket поддерживаются также модели симметричных взаимодействий:

- **запрос/ответ:** поток с одним элементом в запросе и ответе;
- **запрос/поток:** поток с одним элементом в запросе и конечный/бесконечный поток в ответе;

- **запустил и забыл:** поток с одним элементом в запросе и Void (без элементов) поток ответа;
- **канал:** полновесные двунаправленные конечные/бесконечные потоки для запроса и ответа.

Как видите, RSocket предлагает широкий спектр моделей взаимодействия для асинхронной передачи сообщений, создавая только одно соединение в самом начале.

RSocket в Java

Протокол RSocket и модель взаимодействий нашли свое место и востребованы в Java (наряду с реализациями в C++, JS, Python и Go) и реализованы поверх Reactor 3. Модуль RSocket-Java предлагает следующую модель программирования:

```
RSocketFactory                                     // (1)
    .receive()                                     // (1.1)
    .acceptor(new SocketAcceptorImpl())           // (1.2)
    .transport(TcpServerTransport.create("localhost", 7000)) // (1.3)
    .start()                                       // (1.4)
    .subscribe();                                //

RSocket socket = RSocketFactory                   // (2)
    .connect()                                   // (2.1)
    .transport(TcpClientTransport.create("localhost", 7000)) //
    .start()                                     //
    .block();                                    // (2.2)

socket                                             // (3)
    .requestChannel(                             // (3.1)
        Flux.interval(Duration.ofMillis(1000)) //
        .map(i -> DefaultPayload.create("Hello [" + i + "]")) // (3.2)
    )                                             //
    .map(Payload::getDataUtf8)                   // (3.3)
    .doFinally(signalType -> socket.dispose()) //
    .then()                                       //
    .block();                                    //
```

Пояснения к коду.

1. Определение сервера (получателя) RSocket. Наша цель – создать сервер RSocket, поэтому в строке (1.1) вызываем метод `.receive()`. В строке (1.2) подставляем реализацию `SocketAcceptor`, которая определяет метод-обработчик для входящих соединений с клиентами. В строке (1.3) мы определили предпочтительный транспорт, в данном случае TCP. Обратите внимание, что поставщиком транспорта TCP является Reactor-Netty. Наконец, чтобы начать прослушивание определенного адреса, запускаем сервер и вызываем его метод `.subscribe()`.

2. Определение клиента RSocket. В строке (2.1) вместо фабричного метода `.receive()` вызываем `.connect()`, который возвращает клиентский экземпляр RSocket. Обратите внимание, что для простоты в этом примере используется метод `.block()`, чтобы дождаться успешной установки соединения и получить активный экземпляр RSocket.
3. Здесь демонстрируется выполнение запроса к серверу. В этом примере мы используем модель канала (3.1), то есть отправляем и получаем поток. Обратите внимание, что представлением сообщения в потоке по умолчанию является класс `Payload`. Следовательно, в строке (3.2) мы должны завернуть сообщение в экземпляр `Payload` (в данном случае в реализацию по умолчанию `DefaultPayload`) и развернуть его (3.3), например, в строку `String`.

В предыдущем примере мы установили двустороннюю связь между клиентом и сервером. Все взаимодействия здесь выполняются с поддержкой стандарта Reactive Streams и Reactor 3.

Пояснения к коду.

```
class SocketAcceptorImpl implements SocketAcceptor {           // (1)

    @Override                                                  // (2)
    public Mono<RSocket> accept(                                //
        ConnectionSetupPayload setupPayload,                  // (2.1)
        RSocket reactiveSocket                                // (2.2)
    ) {
        return Mono.just(new AbstractRSocket() {              // (3)
            @Override                                           //
            public Flux<Payload> requestChannel(                // (3.1)
                Publisher<Payload> payloads                     // (3.2)
            ) {                                                 //
                return Flux.from(payloads)                      // (3.3)
                    .map(Payload::getDataUtf8)                  //
                    .map(s -> "Echo: " + s)                     //
                    .map(DefaultPayload::create);               //
            }                                                    //
        });                                                    //
    }                                                            //
}                                                                //
```

Вот некоторые пояснения к этому коду.

1. Реализация интерфейса `SocketAcceptor`. Обратите внимание, что `SocketAcceptor` реализует прием соединений на стороне сервера.
2. Интерфейс `SocketAcceptor` имеет только один метод с именем `accept`. Этот метод принимает два параметра, в том числе `ConnectionSetupPayload` (2.1), который представляет первое рукопожатие со стороны клиента во время подключения. Как рассказывалось выше, RSocket по своей природе

является дуплексным соединением. Эта природа представлена вторым параметром `sendingRSocket` метода `асцепт` (2.2). Используя второй параметр, сервер может начать потоковую передачу запросов клиенту, как если бы он был инициатором взаимодействия.

3. Объявление обработчика `RSocket`. В этом случае класс `AbstractRSocket` является абстрактной реализацией интерфейса `RSocket`, которая генерирует исключение `UnsupportedOperationException` для любого метода обработки. Впоследствии, переопределив один из методов (3.1), можем объявить, какие модели взаимодействия поддерживает наш сервер. Наконец, в строке (3.3) предоставляем эхо-функцию, которая принимает текущий поток (3.2) и модифицирует входящие сообщения.

Как видите, определение `SocketAcceptor` не означает определение обработчика. В этом случае вызов метода `SocketAcceptor#асцепт` относится к новому входящему соединению. В свою очередь, интерфейс `RSocket` из `RSocket-Java` представляет клиента и обработчика на стороне сервера одновременно. Наконец, связь между сторонами является одноранговой (*peer-to-peer*), то есть обе стороны могут обрабатывать запросы.

Кроме того, для достижения масштабируемости `RSocket-Java` предлагает модуль `LoadBalancer`, который можно интегрировать с реестром служб, таким как `Eureka`. Например, следующий код демонстрирует упрощенную интеграцию с `Spring Cloud Discovery`.

```
Flux
    .interval(Duration.ofMillis(100))                // (1)
    .map(i ->
        discoveryClient
            .getInstances(serviceId)                  // (2)
            .stream()                                 //
            .map(si ->
                new RSocketSupplier(() ->
                    RSocketFactory.connect()          // (3)
                    .transport(
                        TcpClientTransport.create(
                            si.getHost(),              // (3.1)
                            si.getPort()               // (3.2)
                        )
                    )
                )
            .start()                                 //
        ) {
            public boolean equals(Object obj) { ... } // (4)
                                                    //

            public int hashCode() {                  //
                return si.getUri().hashCode();        // (4.1)
            }                                         //
```



```
        }  
    )  
    .collect(toCollection(ArrayList<RSocketSupplier>::new))  
)  
.as(LoadBalancedRSocketMono::create); // (5)
```

Пояснения к коду.

1. Объявление оператора `.interval()`. Идея состоит в том, чтобы периодически извлекать доступные экземпляры с некоторым идентификатором `serviceId`.
2. Демонстрирует извлечение списка экземпляров.
3. Производитель `Supplier` из `Mono<RSocket>`. Мы используем такие сведения, как имя хоста сервера (3.1) и номер порта (3.2), из каждого `ServiceInstance` для создания соответствующего транспортного соединения.
4. Создание анонимного `RSocketSupplier`. Здесь мы переопределяем методы `equals` и `hashCode`, чтобы иметь возможность выявлять одинаковые экземпляры `RSocketSupplier`. Обратите внимание, что внутренне `LoadBalancedRSocketMono` использует `HashSet`, в котором хранятся все полученные экземпляры. Кроме того, в роли уникального идентификатора экземпляра в группе используем `URI`.
5. Этап преобразования `Flux<Collection<RSocketSupplier>>` в `LoadBalancedRSocketMono`. Отметьте, что хотя результат является экземпляром типа `Mono`, `LoadBalancedRSocketMono` имеет полную информацию о состоянии. Соответственно, каждый новый подписчик потенциально получает разные результаты. Внутренне `LoadBalancedRSocketMono` выбирает экземпляр `RSocket` с использованием прогнозирующего алгоритма балансировки нагрузки и возвращает его подписчику.

В предыдущем примере показан упрощенный метод интеграции `LoadBalancedRSocketMono` с `DiscoveryClient`. Хотя упомянутый пример далеко не оптимален, он достаточно наглядно демонстрирует, как правильно работать с `LoadBalancedRSocketMono`.

Подводя итоги, можно сказать, что `RSocket` – это протокол взаимодействий, который следует семантике Reactive Streams и открывает новые горизонты потоковых взаимодействий через границы сети с поддержкой обратного давления. Кроме того, существует мощная реализация на основе Reactor 3, которая предлагает простой API для создания соединений между одноранговыми узлами и эффективного его использования в течение жизненного цикла взаимодействия.

RSocket и gRPC

У многих из нас все еще может возникать вопрос, зачем использовать отдельный фреймворк, если существует gRPC, который описывается так:

«Высокопроизводительный, с открытым исходным кодом, универсальный фреймворк RPC (<https://github.com/grpc>)».

Этот проект был инициирован компанией Google с целью обеспечить асинхронный обмен сообщениями по HTTP/2. Он использует **Protocol Buffers** (Protobuf) в качестве **языка описания интерфейса** (Interface Description Language, IDL) и в качестве основного формата обмена сообщениями.



Узнать больше об определении служб на языке IDL и Protobuf можно по ссылке <https://grpc.io/docs/guides/concepts.html#service-definition>.

Фреймворк gRPC поддерживает почти ту же семантику сообщений, что и Reactive Streams, и предлагает следующий интерфейс.

```
interface StreamObserver<V> {  
    void onNext(V value);  
    void onError(Throwable t);  
    void onCompleted();  
}
```

Как видите, семантика точно соответствует Observer из RxJava 1. В свою очередь, gRPC API предлагает интерфейс Stream, который расширяет следующие методы:

```
public interface Stream {  
    void request(int numMessages);  
  
    ...  
  
    boolean isReady();  
  
    ...  
}
```

Предыдущий код наглядно показывает, что gRPC наряду с асинхронной передачей сообщений также обеспечивает поддержку управления обратным давлением. Однако эта его часть немного сложнее. Вообще, процесс взаимодействия до определенной степени похож на тот, что мы видели на рис. 8.13. Он отличается лишь возможностью управления процессом с более высокой степенью детализации. Поскольку gRPC построен поверх HTTP/2, фреймворк использует механизм управления HTTP/2 как строительный блок для управления обратным давлением. Но, как бы то ни было, управление все еще зависит от размера скользящего окна в байтах, поэтому управление обратным давлением на уровне логических элементов пока остается нереализованным.

Другое существенное различие между gRPC и RSocket заключается в том, что gRPC – это фреймворк RPC, а RSocket – протокол. gRPC основан на протоколе HTTP/2 и способен автоматически генерировать код с заготовкой будущей службы и клиента. По умолчанию gRPC использует Protobuf как формат сообщений, однако он

поддерживает и другие форматы, такие как JSON. RSocket, в свою очередь, предлагает только реактивную реализацию для сервера и клиента. Кроме того, существует отдельный фреймворк RPC – RSocket-RPC, который основан на протоколе RSocket и предлагает те же возможности, что и gRPC. RSocket-RPC позволяет генерировать код на основе моделей Protobuf, подобно gRPC. Таким образом, любой проект, использующий gRPC, можно без помех перенести на RSocket-RPC.



Узнать больше о RSocket-RPC можно по ссылке <https://github.com/netifi/rsocket-rpc>. Узнать больше о поддержке управления обратным давлением в gRPC можно по ссылке <https://github.com/salesforce/reactive-grpc#back-pressure>.

RSocket в Spring Framework

Хотя реализация предоставляет более широкие возможности для организации асинхронных и высокопроизводительных взаимодействий с низкой задержкой с применением Reactor API, она перекладывает значительную часть работы с конфигурированием инфраструктуры на разработчиков. К счастью, специалисты в Spring оценили данный проект и начали эксперименты по адаптации этого замечательного решения в экосистему Spring посредством упрощенной модели программирования с аннотациями.

Один из экспериментов получил название Spring Cloud Sockets и имел целью предложить знакомую (похожую на Spring Web) модель программирования аннотаций.

```
@SpringBootApplication // (1)
@EnableReactiveSockets // (1.1)
public static class TestApplication { //

    @RequestManyMapping( // (2)
        value = "/stream1", // (2.1)
        mimeType = "application/json" // (2.2)
    ) //
    public Flux<String> stream1(@Payload String a) { // (2.3)
        return Flux.just(a) //
            .mergeWith( //
                Flux.interval(Duration.ofMillis(100)) //
                    .map(i -> "1. Stream Message: [" + i + "]) //
            ); //
    } //
    @RequestManyMapping( // (2)
        value = "/stream2", // (2.1)
        mimeType = "application/json" // (2.2)
    ) //
    public Flux<String> stream2(@Payload String b) { // (2.3)
```

```

        return Flux.just(b)                                //
            .mergeWith(                                    //
                Flux.interval(Duration.ofMillis(500))      //
                    .map(i -> "2. Stream Message: [" + i + "]" ) //
            );                                              //
    }                                                      //
}                                                         //

```

Пояснения к коду.

1. Определение приложения `@SpringBootApplication`. Здесь (1.1) определяем аннотацию `@EnableReactiveSockets`, которая предоставляет необходимые настройки и включает `RSocket` в приложение.
2. Объявление метода-обработчика. Используется аннотация `@RequestManyMapping`, указывающая, что текущий метод работает в модели **запрос/поток**. Модуль `Spring Cloud Sockets` предлагает одну интересную особенность – готовое отображение (маршрутизацию) и позволяет определить путь к обработчику (2.1) и MIME-тип входящих сообщений (2.2). Наконец, имеется дополнительная аннотация `@Payload` (2.3), которая предполагает, что данный параметр является полезной нагрузкой входящего запроса.

Далее представлено уже знакомое приложение `Spring Boot`, в которое при поддержке `Spring Cloud Socket` добавляются дополнительные возможности библиотеки `RSocket-Java`. `Spring Cloud Sockets` упрощает взаимодействие с сервером и с точки зрения клиента.

```

public interface TestClient {                                // (1)
    @RequestManyMapping(                                    // (2)
        value = "/stream1",                                //
        mimeType = "application/json"                      //
    )                                                       //
    Flux<String> receiveStream1(String a);                   //

    @RequestManyMapping(                                    // (2)
        value = "/stream1",                                //
        mimeType = "application/json"                      //
    )                                                       //
    Flux<String> receiveStream2(String b);                   //
}

```

Здесь просто объявили клиента `RSocket` с использованием `Spring Cloud Sockets` (1). Для реализации клиента `RSocket` мы должны определить идентичные аннотации, как в примере с сервером, поверх методов клиента, и соответствующий путь обработчика.

В результате интерфейс легко преобразуется в `Проху` во время выполнения с помощью фабрики `ReactiveSocketClient`.

```
ReactiveSocketClient client = new ReactiveSocketClient(rSocket);
TestClient clientProxy = client.create(TestClient.class);

Flux.merge(
    clientProxy.receiveStream1("a"),
    clientProxy.receiveStream2("b")
)
.log()
.subscribe();
```



Spring Cloud Socket – это экспериментальный проект. В настоящее время он не является официальной частью организации Spring Cloud. Исходный код проекта можно найти в репозитории GitHub: <https://github.com/viniciusccarvalho/spring-cloud-sockets>.

В предыдущем примере мы создали клиента (обратите внимание, что нам пришлось создать экземпляр клиента RSocket вручную). В демонстрационных целях мы объединили два потока и попытались получить результаты вызовом `.log()`.

RSocket в других фреймворках

Как отмечалось выше, модуль Spring Cloud Socket является экспериментальным и больше не поддерживается первоначальным автором. Несмотря на то что команды Spring продолжают внутренние эксперименты и следят за RSocket, поскольку это мощное решение для реактивных потоков через границы Сети, есть несколько других фреймворков, которые также взяли на вооружение реализацию протокола в Java.

Проект ScaleCube

Вот как сами авторы фреймворка описывают проект ScaleCube:

«Проект с открытым исходным кодом, целью которого является упрощение программирования масштабируемых реактивных систем на основе микросервисов (<http://scalecube.io>)».

Главная цель проекта – создание масштабируемых распределенных систем с низкой задержкой.

Для взаимодействий между службами инструментарий использует Project Reactor 3 и в целом не зависит от транспорта. Однако на момент написания этой книги по умолчанию использовался транспорт RSocket-Java.

Наряду с этим ScaleCube предлагает интеграцию с Spring Framework и API на основе аннотаций для построения масштабируемых распределенных систем с низ-

кой задержкой. Не будем вдаваться в подробности интеграции с этим фреймворком здесь. Узнать больше сможете по ссылке <https://github.com/scalecube/scalecube-spring>.

Проект Proteus

Проект Netifi Proteus – еще один мощный набор инструментов. В отличие от Scale Cube, проект Proteus позиционируется как:

«быстрый и простой RPC-слой на основе RSocket для микросервисов (<https://github.com/netifi-proteus>)».

В целом Proteus предлагает облачную платформу для микросервисов, использующую протокол RSocket и фреймворк RSocket-RPC и поддерживающую несколько модулей для маршрутизации, мониторинга и трассировки сообщений.

Кроме того, проект Proteus предлагает интеграцию с Spring Framework и реализует простую модель программирования на основе аннотаций вместе с мощной поддержкой генерации кода. Мы не будем вдаваться в детали интеграции с этим фреймворком – узнать больше сможете по ссылке <https://github.com/netifi-proteus/proteus-spring>.

В заключение о RSocket

Как вы могли заметить в данном разделе, RSocket – это удобный способ построения реактивных систем с высокой пропускной способностью и низкой задержкой, использующих асинхронную одноранговую связь и совместимых со стандартом Reactive Streams. В целом протокол RSocket стремится уменьшить задержку и повысить эффективность системы. Этой цели можно достичь применением неблокирующих взаимодействий через дуплексное соединение. Наряду с этим RSocket предусматривает уменьшение нагрузки на аппаратную часть. Кроме того, RSocket – это протокол, который можно реализовать на любом языке. Наконец, реализация RSocket в Java построена поверх Project Reactor, что обеспечивает мощную модель программирования из коробки.

Сообщество RSocket продолжает стремительно расти и выглядит многообещающе. На данный момент проект активно поддерживают Facebook и Netifi, и в ближайшее время к ним присоединятся другие компании.

Заключение

В этой главе мы пошли по пути преобразования монолитного приложения в реактивную систему. Оценили достоинства и недостатки простых методов баланси-

ровки нагрузки на стороне сервера для создания масштабируемой системы. Однако эти методы не могут обеспечить эластичности, поскольку балансировка на стороне сервера может стать узким местом. Кроме того, данный метод может увеличить затраты на оборудование, потому что требует мощной инфраструктуры для размещения балансировщиков нагрузки.

Также мы рассмотрели методику балансировки нагрузки на стороне клиента. Однако этот метод тоже имеет свои ограничения и не обеспечивает координации с балансировщиками нагрузки на стороне клиента, установленными во всех службах системы.

Наконец, мы рассмотрели, как манифест реактивных систем рекомендует использовать очереди сообщений для надежной асинхронной передачи информации. В результате узнали, что использование брокера сообщений в качестве отдельной реактивной системы для асинхронной связи позволяет достичь эластичности, что делает эту систему полностью реактивной.

Кроме того, мы увидели, как экосистема Spring помогает создавать реактивные системы при поддержке проектов Reactor и Spring Cloud Stream. Познакомились с новыми парадигмами программирования с поддержкой управления обратным давлением такими брокерами сообщений, как Apache Kafka или RabbitMQ. Проработали также несколько примеров применения этого метода на практике.

Затем мы увидели, как Spring Cloud Data Flow помогает отделить конкретную бизнес-логику от конкретных конфигураций, связанных с взаимодействием с брокером сообщений или интеграцией с конкретной облачной платформой.

Наконец, познакомились с дополнительной библиотекой RSocket, обеспечивающей высокопроизводительную связь с низкой задержкой. Мы увидели экспериментальный проект, продемонстрировавший простоту интеграции RSocket в экосистему Spring.

В дополнение к нашим знаниям о реактивности в главе 9 «Тестирование реактивных приложений» исследуем основные методы тестирования реактивных систем, построенных с использованием Spring 5. В заключение посмотрим, как выпускать, поддерживать и контролировать реактивную систему.

Глава 9

Тестирование реактивных приложений

Мы охватили почти все, что касается реактивного программирования с использованием Spring 5.x. Рассмотрели также приемы создания чисто асинхронных реализаций с использованием Project Reactor 3 и применили эти знания для создания веб-приложений с использованием WebFlux. Также мы узнали, как Reactive Spring Data дополняет систему и как быстро можно адаптировать приложение для работы в облачной инфраструктуре, используя Spring Cloud и Spring Cloud Streams.

В этой главе мы дополним наши знания умением тестировать любые компоненты системы. Рассмотрим методы и утилиты тестирования, которые помогут проверить код, написанный с использованием Reactor или другой библиотеки, совместимой со стандартом Reactive Streams. Мы также обсудим функции, предлагаемые Spring Framework для тестирования реактивного приложения от начала до конца.

Будут рассмотрены следующие темы:

- необходимость использования дополнительных инструментов тестирования;
- основы тестирования экземпляров Publisher с помощью StepVerifier;
- продвинутые сценарии использования StepVerifier;
- набор инструментов для сквозного тестирования WebFlux.

Почему реактивные потоки данных сложно тестировать?

Современные корпоративные приложения достигают огромных размеров, потому проверка таких систем является очень важным этапом в любом современном цикле разработки. Однако следует помнить, что в больших системах существует огромное количество компонентов и служб, в которые, в свою очередь, может входить большое количество классов. Чтобы охватить все, мы должны следовать

рекомендациям **Test Pyramid**. Основой тестирования систем, как обычно, является модульное тестирование.

В нашем случае предметом тестирования является код, написанный с использованием приемов реактивного программирования. Как мы видели в главе 3 «*Reactive Streams – новый стандарт потоков*» и в главе 4 «*Project Reactor – основа реактивных приложений*», реактивное программирование дает нам массу преимуществ. Первое из них – возможность оптимизировать использование ресурсов с применением асинхронных взаимодействий. Эта модель хорошо подходит для реализации неблокирующих операций ввода/вывода. Более того, неуклюжий асинхронный код можно преобразовать в чистый и ясный код, используя реактивную библиотеку, например Reactor. Reactor предлагает широкий спектр возможностей, упрощающих разработку приложений.

Однако наряду с преимуществами мы получаем и существенный недостаток – сложность тестирования такого кода. Прежде всего, поскольку код действует асинхронно, нет простого способа взять возвращаемый элемент и убедиться в его правильности. Как рассказывалось в главе 3 «*Reactive Streams – новый стандарт потоков*», мы можем собирать элементы, генерируемые любым издателем Publisher, реализовав интерфейс подписчика Subscriber и используя его для проверки точности полученных результатов. Но решение может получиться очень сложным и малопригодным для тестирования кода.

К счастью, команда Reactor сделала все возможное, чтобы упростить проверку кода, использующего Reactive Streams.

Тестирование реактивных потоков с помощью StepVerifier

Для целей тестирования Reactor предлагает дополнительный модуль reactor-test, который реализует StepVerifier – текучий (fluent) API для построения конвейера тестирования любого издателя Publisher. В следующих разделах расскажем обо всем, что касается модуля **Reactor Test**, начиная с самого необходимого и заканчивая расширенными примерами тестирования.

Основы StepVerifier

Для проверки издателя Publisher используются два основных метода. Первый из них – StepVerifier.<T>create(Publisher<T> source). Вот как может выглядеть конструкция теста, реализованного с применением этого метода:

```
StepVerifier
    .create(Flux.just("foo", "bar"))
```

```
.expectSubscription()  
.expectNext("foo")  
.expectNext("bar")  
.expectComplete()  
.verify();
```

В этом примере наш издатель `Publisher` должен создать два конкретных элемента, а последующие операции проверяют, были ли эти элементы доставлены конечному подписчику. Этот пример достаточно наглядно показывает работу `StepVerifier`. Принцип использования построителя, предлагаемый этим классом, позволяет определить порядок, в котором будут происходить события в процессе проверки. Согласно предыдущему коду, первое отправленное событие должно быть событием, относящимся к подписке, а следующие события должны быть строками "foo" и "bar". И наконец, `StepVerifier#waitCompletion` определяет наличие сигнала завершения, который в нашем случае должен быть вызовом `Subscriber#onComplete` или просто успешным завершением данного потока. Чтобы выполнить проверку или, другими словами, выполнить подписку, дабы инициировать процесс, нужно вызвать метод `.verify()`. Это блокирующий вызов, поэтому он блокирует выполнение до тех пор, пока поток не отправит все ожидаемые события.

Используя эту простую методику, можно проверить `Publisher` с исчисляемым количеством элементов и событий. Но проверить поток с огромным количеством элементов будет очень трудно. Когда более важно проверить, сгенерировал ли издатель конкретное количество элементов, а не конкретные значения, можно использовать метод `.expectNextCount()`, как показано ниже.

```
StepVerifier  
    .create(Flux.range(0, 100))  
    .expectSubscription()  
    .expectNext(0)  
    .expectNextCount(98)  
    .expectNext(99)  
    .expectComplete()  
    .verify();
```

Как мы уже знаем, `Flux.range(0, 100)` создает диапазон элементов от 0 до 99 включительно. В таких случаях важнее проверить, было ли послано это конкретное количество элементов и посылались ли, например, элементы в правильном порядке. Выполнить такую проверку можно с помощью пары методов – `.expectNext()` и `.expectNextCount()`. Как демонстрирует предыдущий пример, первый элемент будет проверен оператором `.expectNext(0)`. Затем тестовый конвейер проверит отправку издателем еще 98 элементов, то есть проверит, отправил ли данный издатель в общей сложности 99 элементов. Поскольку наш издатель должен создать 100 элементов, последний элемент должен быть 99, что проверяется с помощью оператора `.expectNext(99)`.

Несмотря на то что метод `.expectNextCount()` решает часть проблемы, иногда недостаточно просто проверить количество посылаемых элементов. Например, когда дело доходит до проверки кода, который отвечает за фильтрацию или выбор элементов по определенным правилам, важно убедиться, что все отправленные элементы соответствуют определенному правилу фильтрации. Для этого `StepVerifier` позволяет одновременно записывать отправленные данные и их проверку, используя такие инструменты, как `Java Hamcrest` (см. <http://hamcrest.org/JavaHamcrest>). В следующем примере демонстрируется модульный тест, который использует эту библиотеку.

```
Publisher<Wallet> usersWallets = findAllUsersWallets();
StepVerifier
    .create(usersWallets)
    .expectSubscription()
    .recordWith(ArrayList::new)
    .expectNextCount(1)
    .consumeRecordedWith(wallets -> assertThat(
        wallets,
        everyItem(hasProperty("owner", equalTo("admin"))))
    )
    .expectComplete()
    .verify();
```

Рассматривая предыдущий пример, можно понять, как записать все элементы, а затем сопоставить их с заданным условием. В отличие от предыдущих примеров, где каждая проверка охватывала только один или указанное количество элементов, `.consumeRecordedWith()` позволяет проверить все элементы, которые публикуются данным издателем `Publisher`. Следует отметить, что `.consumeRecordedWith()` работает только в паре с `.recordWith()`. Мы также должны тщательно определить класс коллекции, в которой будут храниться записи. В случае с многопоточным издателем тип коллекции, используемой для записи событий, должен поддерживать параллельный доступ, поэтому в таких случаях лучше использовать `.recordWith(ConcurrentLinkedQueue::new)` вместо `.recordWith(ArrayList::new)`, потому что `ConcurrentLinkedQueue` является потокобезопасным, в отличие от `ArrayList`.

В предыдущих абзацах мы познакомились с основами `Reactor Test API`. Однако существуют и другие похожие методы. Например, прием и проверку следующего элемента можно определить так:

```
StepVerifier
    .create(Flux.just("alpha-foo", "beta-bar"))
    .expectSubscription()
    .expectNextMatches(e -> e.startsWith("alpha"))
    .expectNextMatches(e -> e.startsWith("beta"))
    .expectComplete()
    .verify();
```

Единственное различие между `.expectNextMatches()` и `.expectNext()` состоит в том, что первый позволяет определить свой предикат `Predicate`. Это различие объясняется тем, что `.expectNext()` основан на сравнении элемента с использованием метода `.equals()`.

Аналогично методы `.assertNext()` и `.consumeNextWith()` позволяют писать собственные утверждения. Следует отметить, что `.assertNext()` является псевдонимом для `.consumeNextWith()`. Разница между `.expectNextMatches()` и `.assertNext()` заключается в том, что первый принимает предикат `Predicate`, который должен возвращать `true` или `false`, а последний принимает потребителя `Consumer`, который может вызвать исключение, и любая ошибка `AssertionError`, сгенерированная потребителем, будет перехвачена методом `.verify()`, как показано в следующем фрагменте:

```
StepVerifier
    .create(findUsersUSDWallet())
    .expectSubscription()
    .assertNext(wallet -> assertThat(
        wallet,
        hasProperty("currency", equalTo("USD")))
    )
    .expectComplete()
    .verify();
```

Наконец, у нас остаются неохваченными ошибочные ситуации, которые также являются частью нормального жизненного цикла системы. В API существует несколько методов, которые позволяют проверять сигналы ошибок. Самый простой из них – `.expectError()` без аргументов, как показано ниже.

```
StepVerifier
    .create(Flux.error(new RuntimeException("Error")))
    .expectError()
    .verify();
```

Мы, конечно, можем проверить, не появилась ли ошибка, но иногда жизненно важно определить конкретный тип ошибки. Например, если при входе пользователя в систему введены неверные учетные данные, служба безопасности должна послать ошибку `BadCredentialsException.class`. Чтобы проверить ее, можно использовать `.expectError (Class <? Extends Throwable>)`, как показано ниже.

```
StepVerifier
    .create(securityService.login("admin", "wrong"))
    .expectSubscription()
    .expectError(BadCredentialsException.class)
    .verify();
```

В тех случаях, когда проверки типа ошибки недостаточно, существуют дополнительные расширения `.expectErrorMatches()` и `.consumeErrorWith()`, которые позволяют прямое взаимодействие с `Throwable`.

Теперь у нас есть некоторое представление об основах тестирования кода, написанного с использованием `Reactor 3` или любой библиотеки, совместимой со стандартом `Reactive Streams`. `StepVerifier` API охватывает большинство реактивных рабочих процессов. Однако в условиях реальной разработки порой могут возникать некоторые необычные ситуации.

Продвинутые приемы тестирования с использованием `StepVerifier`

Первым этапом тестирования издателя `Publisher` является проверка бесконечного потока. Согласно стандарту `Reactive Streams` это означает, что издатель никогда не вызовет метод `Subscriber#onComplete()`. Это также означает, что методы тестирования, с которыми мы познакомились выше, больше не будут работать. Проблема в том, что `StepVerifier` будет бесконечно ждать сигнала завершения, а значит, тест останется заблокированным, пока не будет принудительно остановлен. Для устранения этой проблемы `StepVerifier` предлагает API, который аннулирует подписку после удовлетворения некоторых ожиданий, как показано в следующем коде:

```
Flux<String> websocketPublisher = ...
StepVerifier
    .create(websocketPublisher)
    .expectSubscription()
    .expectNext("Connected")
    .expectNext("Price: $12.00")
    .thenCancel()
    .verify();
```

Этот код отключается, или отписывается, от `WebSocket` после получения сообщений `Connected` и `Price: $12.00`.

Другой важный этап в тестировании системы – проверка реакции издателя на обратное давление. Например, при взаимодействии с внешней системой через `WebSocket` мы получаем чистую модель *PUSH* для издателя. Простейший способ предотвращения такого поведения – защита нисходящего потока с помощью оператора `.onBackpressureBuffer()`. Чтобы проверить, работает ли система в соответствии с выбранной стратегией обратного давления, мы должны вручную контролировать потребности подписчика. Для этого `StepVerifier` предлагает метод `.thenRequest()`, как демонстрирует следующий код:

```

Flux<String> websocketPublisher = ...
Class<Exception> expectedErrorClass =
    reactor.core.Exceptions.failWithOverflow().getClass();

StepVerifier
    .create(websocketPublisher.onBackpressureBuffer(5), 0)
    .expectSubscription()
    .thenRequest(1)
    .expectNext("Connected")
    .thenRequest(1)
    .expectNext("Price: $12.00")
    .expectError(expectedErrorClass)
    .verify();

```

Предыдущий пример показывает, как использовать метод `.thenRequest()` для проверки реакции на обратное давление. Как ожидается, в какой-то момент времени произойдет переполнение, и мы получим ошибку переполнения. Обратите внимание, что здесь мы использовали перегруженную версию метода `StepVerifier.create()`, которая принимает начальную потребность подписчика во втором аргументе. В версии метода с одним аргументом потребность получает значение по умолчанию `Long.MAX_VALUE`, означающее неограниченную потребность.

Одной из продвинутых возможностей, предлагаемых `StepVerifiers`, является возможность выполнения дополнительного действия после определенной проверки. Например, если процесс, генерирующий элементы, требует некоторого дополнительного внешнего взаимодействия, это можно реализовать с помощью метода `.then()`.

Также можно использовать `TestPublisher` из пакета с инструментами тестирования в библиотеке `Reactor`. `TestPublisher` реализует интерфейс `Publisher` и позволяет непосредственно вызывать `onNext()`, `onComplete()` и `onError()` в процессе тестирования. Следующий пример демонстрирует, как генерировать новые события в ходе тестирования.

```

TestPublisher<String> idsPublisher = TestPublisher.create();

StepVerifier
    .create(walletsRepository.findAllById(idsPublisher))
    .expectSubscription()
    .then(() -> idsPublisher.next("1")) // (1)
    .assertNext(w -> assertThat(w, hasProperty("id", equalTo("1")))) // (2)
    .then(() -> idsPublisher.next("2")) // (3)
    .assertNext(w -> assertThat(w, hasProperty("id", equalTo("2")))) // (4)
    .then(idsPublisher::complete) // (5)
    .expectComplete()
    .verify();

```

В этом примере необходимо убедиться, что `WalletRepository` отыщет кошельки с заданными идентификаторами. Одно из особых требований к хранилищу кошельков – возможность поиска по мере поступления данных, а это означает, что входящий поток должен быть горячим издателем (то есть посылать данные независимо от наличия подписчиков). В нашем примере мы используем `TestPublisher.next()` в сочетании с `StepVerifier.then()`. Шаги (1) и (3) посылают новые запросы только после проверки предыдущих шагов. Шаг (2) проверяет успешность обработки запросов, сгенерированных на шаге (1), а шаг (4) проверяет шаг (3). Шаг (5) дает команду `TestPublisher` завершить поток запросов, после чего `StepVerifier` убеждается, что поток ответов также завершился.

Эта методика играет важную роль, позволяя генерировать события после фактической подписки. Таким образом, мы можем убедиться, что посланные идентификаторы были найдены сразу после этого и что `walletsRepository` действует так, как ожидалось.

Виртуальное время

Главной целью тестирования является бизнес-логика, но есть еще одна очень важная часть, о которой следует подумать. Чтобы понять, что это за часть, рассмотрим следующий пример:

```
public Flux<String> sendWithInterval() {
    return Flux.interval(Duration.ofMinutes(1))
        .zipWith(Flux.just("a", "b", "c"))
        .map(Tuple2::getT2);
}
```

В примере показан наивный способ публикации событий с определенным интервалом. В действующем приложении за тем же API может быть скрыт более сложный механизм, включающий большие задержки, тайм-ауты и интервалы событий. Проверить такой код с помощью `StepVerifier` можно так, как показано ниже.

```
StepVerifier
    .create(sendWithInterval())
    .expectSubscription()
    .expectNext("a", "b", "c")
    .expectComplete()
    .verify();
```

Тест будет пройден нашей предыдущей реализацией `sendWithInterval()`, чего мы действительно хотим достичь. Однако у теста есть проблема. Если запустить его несколько раз, обнаружится, что средняя продолжительность теста составляет чуть больше трех минут. Это объясняется тем, что метод `sendWithInterval()` создает три события с задержкой в одну минуту перед каждым элементом. В случаях

когда интервалы измеряются часами или даже днями, тестирование системы может занять огромное количество времени, что недопустимо, если речь идет о непрерывной интеграции. Для решения этой проблемы модуль Reactor Test предлагает возможность замены реального времени виртуальным, как показано в следующем коде:

```
StepVerifier.withVirtualTime() -> sendWithInterval()  
// сценарий тестирования ...
```

Используя метод-построитель `.withVirtualTime()`, мы явно замещаем каждый планировщик Scheduler из Reactor реализацией `reactor.test.scheduler.VirtualTimeScheduler`. Это означает, что `Flux.interval` также будет действовать в данном планировщике Scheduler. Соответственно, появляется возможность управлять временем с использованием `VirtualTimeScheduler#advanceTimeBy`, как показано ниже.

```
StepVerifier  
    .withVirtualTime() -> sendWithInterval()  
    .expectSubscription()  
    .then(() -> VirtualTimeScheduler  
        .get()  
        .advanceTimeBy(Duration.ofMinutes(3))  
    )  
    .expectNext("a", "b", "c")  
    .expectComplete()  
    .verify();
```

В предыдущем примере мы используем `.then()` в сочетании с `VirtualTimeScheduler` для ускорения течения времени на определенную величину. Если мы запустим этот тест, он займет несколько миллисекунд вместо минут! Этот результат намного лучше, поскольку теперь наш тест ведет себя независимо от фактических временных интервалов, с которыми производятся данные. Наконец, чтобы сделать тест более ясным, можно заменить комбинацию `.then()` и `VirtualTimeScheduler` на `.thenAwait()`, который действует точно так же.



Имейте в виду, что если `StepVerifier` недостаточно ускоряет время, тест может зависнуть.

Чтобы ограничить время, затрачиваемое на тестирование, можно использовать перегруженную версию `.verify(Duration t)`. Она вызовет `AssertionError`, когда тест не сможет выполнить проверку в течение разрешенного периода времени. Кроме того, метод `.verify()` возвращает время, которое фактически занял процесс проверки. Следующий код описывает такой вариант использования:

```
Duration took = StepVerifier  
    .withVirtualTime() -> sendWithInterval()  
    .expectSubscription()
```



```
.thenAwait(Duration.ofMinutes(3))  
.expectNext("a", "b", "c")  
.expectComplete()  
.verify();
```

```
System.out.println("Verification took: " + took);
```

В ситуациях, когда важно убедиться, что в течение указанного времени не было никаких событий, можно использовать дополнительный метод `.expectNoEvents()`. Используя его, можно проверить, генерируются ли события через заданный интервал, как показано ниже.

```
StepVerifier  
    .withVirtualTime(() -> sendWithInterval())  
    .expectSubscription()  
    .expectNoEvent(Duration.ofMinutes(1))  
    .expectNext("a")  
    .expectNoEvent(Duration.ofMinutes(1))  
    .expectNext("b")  
    .expectNoEvent(Duration.ofMinutes(1))  
    .expectNext("c")  
    .expectComplete()  
    .verify();
```

Предыдущие примеры наглядно демонстрируют приемы, помогающие ускорить тестирование.

Обратите внимание, что существует дополнительная перегруженная версия метода `.thenAwait()` без аргументов. Основная идея этого метода – запуск любых задач, которые еще не выполнены и которые планируется выполнить в текущее виртуальное время или до него. Например, чтобы получить первое запланированное событие в конфигурации `Flux.interval(Duration.ofMillis(0), Duration.ofMillis(1000))`, потребуется дополнительно вызвать `.thenAwait()`, как показано в следующем коде.

```
StepVerifier  
    .withVirtualTime(() ->  
        Flux.interval(Duration.ofMillis(0), Duration.ofMillis(1000))  
            .zipWith(Flux.just("a", "b", "c"))  
            .map(Tuple2::getT2)  
        )  
    .expectSubscription()  
    .thenAwait()  
    .expectNext("a")  
    .expectNoEvent(Duration.ofMillis(1000))  
    .expectNext("b")  
    .expectNoEvent(Duration.ofMillis(1000))
```

```
.expectNext("c")  
.expectComplete()  
.verify();
```

Без `.thenAwait()` тест зависнет.

Проверка реактивного контекста

Наконец, самая необычная проверка – это проверка контекста `Context` из `Reactor`. Мы рассмотрели роль и механику работы контекста в главе 4 «*Project Reactor – основа реактивных приложений*». Предположим, мы хотим проверить реактивный API службы аутентификации. Для аутентификации пользователя `LoginService` ожидает, что подписчик предоставит экземпляр `Context` с информацией для аутентификации.

```
StepVerifier  
    .create(securityService.login("admin", "admin"))  
    .expectSubscription()  
    .expectAccessibleContext()  
    .hasKey("security")  
    .then()  
    .expectComplete()  
    .verify();
```

Этот пример показывает, как проверить существование экземпляра `Context`. Как видите, есть только один случай, когда проверка `.expectAccessibleContext()` может завершиться неудачно. Это происходит, только когда возвращаемый издатель `Publisher` не является типом `Reactor` (`Flux` или `Mono`). Поэтому последующие проверки контекста выполняются только при наличии доступного контекста. Наряду с `.hasKey()` существует множество других методов, которые позволяют детально проверить текущий контекст. Чтобы выйти из проверки контекста, конструктор предоставляет метод `.then()`.

Итак, в этом разделе мы узнали, как `Reactor Test` помогает в тестировании реактивных потоков. Раздел охватывает почти все, что требуется для модульного тестирования небольшого фрагмента реактивного кода, но вообще `Spring Framework` предлагает гораздо больше возможностей для тестирования реактивных систем.

Тестирование WebFlux

В этом разделе рассмотрим дополнительные возможности проверки приложений на основе `WebFlux`. Сосредоточимся на проверке совместимости модулей, целостности приложений, поддержке открытых протоколов связи, внешних API и клиентских библиотек. То есть речь идет уже не о простых модульных тестах, а о компонентном и интеграционном тестировании.

Тестирование контроллеров с помощью WebTestClient

Представьте, что мы тестируем *службу платежей*. Предположим, эта служба поддерживает методы GET и POST для конечной точки /payment. Первый HTTP-вызов извлекает список выполненных платежей для текущего пользователя. Второй дает возможность выполнить новый платеж. Реализация соответствующего REST-контроллера выглядит следующим образом:

```
@RestController
@RequestMapping("/payments")
public class PaymentController {
    private final PaymentService paymentService;

    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @GetMapping("/")
    public Flux<Payment> list() {
        return paymentService.list();
    }

    @PostMapping("/")
    public Mono<String> send(Mono<Payment> payment) {
        return paymentService.send(payment);
    }
}
```

Первый шаг в процессе проверки службы – запись всех ожиданий от взаимодействий со службой. Для взаимодействия с конечными точками WebFlux обновленный модуль spring-test предлагает новый класс org.springframework.test.web.reactive.server.WebTestClient. WebTestClient похож на org.springframework.test.web.servlet.MockMvc. Единственное отличие состоит в том, что WebTestClient предназначен для тестирования конечных точек WebFlux. Например, используя WebTestClient и библиотеку Mockito, можем проверить получение списка пользовательских платежей, как показано ниже.

```
@Test
public void verifyRespondWithExpectedPayments() {
    PaymentService paymentService = Mockito.mock(PaymentService.class);
    PaymentController controller = new PaymentController(paymentService);

    prepareMockResponse(paymentService);
    WebTestClient
        .bindToController(controller)
```

```
.build()
.get()
.uri("/payments/")
.exchange()
.expectHeader().contentTypeCompatibleWith(APPLICATION_JSON)
.expectStatus().is2xxSuccessful()
.returnResult(Payment.class)
.getResponseBody()
.as(StepVerifier::create)
.expectNextCount(5)
.expectComplete()
.verify();
}
```

В этом примере мы реализовали проверку `PaymentController` с помощью `WebTestClient`. Используя текущий API класса `WebTestClient`, можем проверить код состояния и заголовки ответа. Также можем использовать `.getResponseBody()`, чтобы получить поток Flux ответов, который в конечном итоге проверяется с помощью `StepVerifier`. Данный пример показывает, насколько легко оба инструмента могут интегрироваться друг с другом.

В предыдущем примере можно заметить, что `PaymentService` является фиктивной службой, и мы не взаимодействуем с внешними службами при тестировании `PaymentController`. Однако, чтобы убедиться в целостности системы, мы должны запустить полноценные компоненты, а не только несколько уровней. Чтобы выполнить нормальное интеграционное тестирование, нужно запустить все приложение. Для этой цели можно использовать аннотацию `@SpringBootTest` в сочетании с `@AutoConfigureWebTestClient`. `WebTestClient` предоставляет возможность установить HTTP-соединение с любым HTTP-сервером. Кроме того, `WebTestClient` может напрямую связываться с приложениями на основе `WebFlux`, используя фиктивные объекты запросов и ответов, что устраняет необходимость в HTTP-сервере. Он играет ту же роль в тестировании приложений `WebFlux`, что и `TestRestTemplate` для приложений `WebMVC`, как показано в следующем коде.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWebTestClient
public class PaymentControllerTests {
    @Autowired
    WebTestClient client;
    ...
}
```

Здесь нам больше не нужно настраивать `WebTestClient`. Это связано с тем, что все необходимые значения по умолчанию настраиваются с помощью механизма автоматической настройки Spring Boot.



Обратите внимание, что если приложение использует модуль Spring Security, может потребоваться дополнительная настройка для тестов. Мы можем добавить зависимость от модуля `spring-security-test`, который предоставляет аннотацию `@WithMockUser`, предназначенную, в частности, для проверки подлинности пользователя. Изначально механизм `@WithMockUser` поддерживает `WebTestClient`. Однако даже если `@WithMockUser` выполняет свою работу, включенная по умолчанию функция CSRF может добавить некоторые нежелательные препятствия для сквозных тестов. Отметьте также, что дополнительная настройка, касающаяся CSRF, требуется только для `@SpringBootTest` или любого другого механизма выполнения тестов Spring Boot, кроме `@WebFluxTest`, которая по умолчанию отключает CSRF.

Чтобы протестировать вторую часть службы платежей, нужно проверить бизнес-логику реализации `PaymentService`, как показано ниже.

```
@Service
public class DefaultPaymentService implements PaymentService {

    private final PaymentRepository paymentRepository;
    private final WebClient client;

    public DefaultPaymentService(PaymentRepository repository,
                                WebClient.Builder builder) {
        this.paymentRepository = repository;
        this.client = builder.baseUrl("http://api.bank.com/submit").build();
    }

    @Override
    public Mono<String> send(Mono<Payment> payment) {
        return payment
            .zipWith(
                ReactiveSecurityContextHolder.getContext(),
                (p, c) -> p.withUser(c.getAuthentication().getName())
            )
            .flatMap(p -> client
                .post()
                .syncBody(p)
                .retrieve()
                .bodyToMono(String.class)
                .then(paymentRepository.save(p)))
            .map(Payment::getId);
    }

    @Override
    public Flux<Payment> list() {
        return ReactiveSecurityContextHolder
```

```
.getContext()
.map(SecurityContext::getAuthentication)
.map(Principal::getName)
.flatMapMany(paymentRepository::findAllByUser);
    }
}
```

Прежде всего отметим, что метод, возвращающий все платежи пользователя, взаимодействует только с базой данных. Напротив, логика отправки платежа наряду со взаимодействием с базой данных требует дополнительного взаимодействия с внешней системой через `WebClient`. В нашем примере мы используем реактивный модуль `Spring Data MongoDB`, который поддерживает встроенный режим тестирования. Взаимодействие с внешним банковским провайдером, напротив, не может быть встроенным. Следовательно, нам нужно смоделировать внешнюю службу с помощью такого инструмента, как `WireMock` (<http://wiremock.org>), или каким-то образом симитировать исходящие HTTP-запросы. Фиктивные службы, реализованные с применением `WireMock`, являются допустимым вариантом как для `WebMVC`, так и для `WebFlux`.

Однако при сравнении `WebMVC` и `WebFlux` с точки зрения тестирования первый имеет преимущество, потому что «из коробки» позволяет имитировать исходящее HTTP-взаимодействие. К сожалению, в `Spring Boot 2.0` и `Spring Framework 5.0.x` отсутствует аналогичная возможность для `WebClient`. Однако есть один трюк, который позволяет симитировать ответ на исходящий HTTP-вызов. В ситуациях, когда разработчики следуют методике построения `WebClient` с использованием `WebClient.Builder`, можно смоделировать `org.springframework.web.reactive.function.client.ExchangeFunction`, играющую важную роль в обработке запросов `WebClient`, как показано ниже.

```
public interface ExchangeFunction {
    Mono<ClientResponse> exchange(ClientRequest request);
    ...
}
```

Используя следующую конфигурацию, можно настроить `WebClient.Builder` и представить фиктивную реализацию `ExchangeFunction`.

```
@TestConfiguration
public class TestWebClientBuilderConfiguration {
    @Bean
    public WebClientCustomizer testWebClientCustomizer(
        ExchangeFunction exchangeFunction
    ) {
        return builder -> builder.exchangeFunction(exchangeFunction);
    }
}
```

Этот трюк дает возможность проверить правильность сформированного Client Request. Кроме того, правильно реализовав ClientResponse, можно смоделировать сетевую активность и взаимодействие с внешней службой. Законченный тест мог бы выглядеть примерно так:

```
@ImportAutoConfiguration({
    TestSecurityConfiguration.class,
    TestWebClientBuilderConfiguration.class
})
@RunWith(SpringRunner.class)
@WebFluxTest
@AutoConfigureWebTestClient
public class PaymentControllerTests {
    @Autowired
    WebTestClient client;
    @MockBean
    ExchangeFunction exchangeFunction;
    @Test
    @WithMockUser
    public void verifyPaymentsWasSentAndStored() {
        Mockito
            .when(exchangeFunction.exchange(Mockito.any()))
            .thenReturn(
                Mono.just(MockClientResponse.create(201, Mono.empty())));
        client.post()
            .uri("/payments/")
            .syncBody(new Payment())
            .exchange()
            .expectStatus().is2xxSuccessful()
            .returnResult(String.class)
            .getResponseBody()
            .as(StepVerifier::create)
            .expectNextCount(1)
            .expectComplete()
            .verify();

        Mockito.verify(exchangeFunction).exchange(Mockito.any());
    }
}
```

В этом примере мы использовали аннотацию `@WebFluxTest`, чтобы отключить полностью автоматическую настройку и применить только необходимые настройки WebFlux, включая `WebTestClient`. Аннотация `@MockBean` используется для внедрения фиктивного экземпляра `ExchangeFunction` в контейнер Spring IoC. Сочетание `Mockito` и `WebTestClient` позволяет построить сквозную проверку желаемой бизнес-логики.

В приложениях WebFlux можно имитировать исходящие HTTP-взаимодействия по аналогии WebMVC, но используйте эту возможность с большой осторожностью! Данный подход имеет свои подводные камни. В настоящее время тестирование приложений строится в предположении, что все HTTP-взаимодействия реализованы с помощью WebClient. Это не контракт на обслуживание, а скорее деталь реализации. Следовательно, если какая-либо служба по какой-либо причине сменит клиентскую библиотеку поддержки HTTP, соответствующие тесты перестанут выполняться без всякой причины. Следовательно, для имитации внешних служб предпочтительнее использовать WireMock. Такой подход не только предполагает использование конкретной клиентской библиотеки HTTP, но и тестирует фактическую полезную нагрузку в запросах/ответах, отправляемых по Сети. Как показывает практика, при тестировании отдельных классов с помощью бизнес-логики допустимо использовать клиентскую библиотеку HTTP, но этого абсолютно не следует делать для тестирования всей службы.

Вообще, мы можем проверить всю бизнес-логику, построенную на основе стандартного Spring WebFlux API. WebTestClient имеет выразительный API, позволяющий проверять обычные REST-контроллеры и новые функции маршрутизации с использованием `.bindToRouterFunction()` или `.bindToWebHandler()`. Более того, с помощью WebTestClient можно выполнять тестирование «черного ящика», используя `.bindToServer()` и предоставляя полный HTTP-адрес сервера. Следующий тест проверяет некоторые предположения относительно веб-сайта `http://www.bbc.com` и терпит неудачу (как и ожидалось) из-за разницы между ожидаемым и фактическим ответами.

```
WebTestClient webTestClient = WebTestClient
    .bindToServer()
    .baseUrl("http://www.bbc.com")
    .build();

webTestClient
    .get()
    .exchange()
    .expectStatus().is2xxSuccessful()
    .expectHeader().exists("ETag")
    .expectBody().json("{}");
```

Этот пример показывает, что классы WebClient и WebTestClient из WebFlux реализуют не только асинхронных и неблокирующих HTTP-клиентов, но и текущий API для интеграционного тестирования.

Тестирование WebSocket

Наконец, мы должны рассмотреть еще одну тему – тестирование потоковых систем. В этом разделе рассмотрим тестирование только сервера и клиента

WebSocket. Как рассказывалось в главе 6 «Неблокирующие и асинхронные взаимодействия с *WebFlux*», кроме WebSocket API существует также протокол **Server-Sent Events** (SSE) для потоковой передачи данных, который предлагает аналогичные возможности. Однако, поскольку реализация SSE практически идентична реализации обычного контроллера, все методы тестирования из предыдущего раздела будут действительны и для этого случая. Следовательно, единственное, что пока остается неясным, – это то, как тестировать WebSocket.

К сожалению, WebFlux не предлагает готового решения для тестирования WebSocket API. Однако для создания тестовых классов можно использовать стандартный набор инструментов. В частности, для подключения к целевому серверу и проверки полученных данных можно использовать `WebSocketClient`. Этот подход демонстрируется ниже.

```
new ReactorNettyWebSocketClient()  
    .execute(uri, new WebSocketHandler() {...})
```

Конечно, мы можем подключиться к серверу, но очень сложно проверить входящие данные с помощью `StepVerifier`. Прежде всего `.execute()` возвращает `Mono<Void>` вместо входящих данных. Кроме того, мы должны проверить двусторонние взаимодействия, а значит, иногда важно убедиться, что входящие данные являются результатом исходящих сообщений. Примером такой системы может служить торговая платформа. Предположим, у нас есть платформа торговли криптовалютами, которая предлагает возможность отправлять сделки и получать результаты сделок. Одним из требований бизнеса является возможность совершать сделки с биткойнами. Это означает, что пользователь может продавать или покупать биткойны и наблюдать за результатом сделки. Проверяя функциональность, мы должны убедиться, что входящая сделка является результатом исходящего запроса. С точки зрения тестирования трудно иметь дело с `WebSocketHandler` для проверки всех краевых случаев. Следовательно, с точки зрения тестирования интерфейс клиента WebSocket в идеале должен выглядеть следующим образом:

```
interface TestWebSocketClient {  
    Flux<WebSocketMessage> sendAndReceive(Publisher<?> outgoingSource);  
}
```

Чтобы адаптировать `WebSocketClient` к предлагаемому `TestWebSocketClient`, нужно выполнить следующие шаги.

Во-первых, следует предусмотреть обработку `WebSocketSession` в `WebSocketHandler` через `Mono<WebSocketSession>`, как показано ниже:

```
Mono.create(sink ->  
    sink.onCancel(  
        client.execute(uri, session -> {  
            sink.success(session);  
            return Mono.never();  
        })
```

```

    })
    .doOnError(sink::error)
    .subscribe()
  )
};

```

Благодаря `Mono.create()` и `MonoSink` можем использовать старый способ обработки асинхронных обратных вызовов с сеансом и перенаправить его в другой поток. В свою очередь, нам нужно позаботиться о правильном типе возвращаемого значения метода `WebSocketHandler#handle`. Это связано с тем, что тип возвращаемого значения контролирует время жизни открытого соединения. С другой стороны, соединение должно быть закрыто, как только `MonoSink` уведомит нас об этом. Следовательно, `Mono.never()` является лучшим кандидатом, который в сочетании с ошибкой перенаправления с помощью `.doOnError(sink::error)` и отменой обработки с помощью `sink.onCancel()` делает эту адаптацию завершённой.

Второе, что нужно сделать в отношении предлагаемого API, – адаптировать `WebSocketSession` с помощью следующего приема.

```

public Flux<WebSocketMessage> sendAndReceive(
    Publisher<?> outgoingSource
) {
    ...
    .flatMapMany(session ->
        session.receive()
            .mergeWith(
                Flux.from(outgoingSource)
                    .map(Object::toString)
                    .map(session::textMessage)
                    .as(session::send)
                    .then(Mono.empty())
            )
    );
}

```

Здесь мы посылаем входящие сообщения `WebSocketMessage` на сервер. Как видите, в этом примере используется простое преобразование объекта, которое можно заменить сложным отображением сообщений.

Наконец, используя этот API, можно построить следующий процесс для проверки упомянутой функциональности.

```

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment =
    SpringBootTest.WebEnvironment.DEFINED_PORT)
public class WebSocketAPITests {

```

```
@Test
@WithMockUser
public void checkThatUserIsAbleToMakeATrade() {
    URI uri = URI.create("ws://localhost:8080/stream");
    TestWebSocketClient client = TestWebSocketClient.create(uri);
    TestPublisher<String> testPublisher = TestPublisher.create();
    Flux<String> inbound = testPublisher
        .flux()
        .subscribeWith(ReplayProcessor.create(1))
        .transform(client::sendAndReceive)
        .map(WebSocketMessage::getPayloadAsText);

    StepVerifier
        .create(inbound)
        .expectSubscription()
        .then(() -> testPublisher.next("TRADES|BTC"))
        .expectNext("PRICE|AMOUNT|CURRENCY")
        .then(() -> testPublisher.next("TRADE: 10123|1.54|BTC"))
        .expectNext("10123|1.54|BTC")
        .then(() -> testPublisher.next("TRADE: 10090|-0.01|BTC"))
        .expectNext("10090|-0.01|BTC")
        .thenCancel()
        .verify();
}
```

Первое, что делается в этом примере, – настраивается `WebEnvironment`. Устанавливая `WebEnvironment.DEFINED_PORT`, мы сообщаем Spring Framework, что он должен прослушивать указанный порт. Это важный шаг, потому что `WebSocketClient` может подключиться к определенному обработчику только через реальный HTTP-вызов. Затем готовится входящий поток. В данном случае важно кешировать первое сообщение, отправленное через `TestPublisher` в шаге `.then()`, потому что `.then()` может быть вызван до получения сеанса, то есть первое сообщение может быть проигнорировано и мы можем не подключиться к нашим сделкам с биткойнами. Следующий шаг – проверка прохождения отправленных сделок и получения правильного ответа.

Наконец, следует упомянуть, что наряду с проверкой `WebSocket API` иногда бывает необходимо симитировать взаимодействие через `WebSocketClient` с внешними службами. К сожалению, не существует простого способа сделать это, прежде всего из-за отсутствия универсального `WebSocketClient.Build`, который можно было бы симитировать. Не существует готового способа автоматически подключить `WebSocketClient`. Следовательно, единственное решение, которое у нас остается, – это подключение к фиктивному серверу.

Заключение

В этой главе мы узнали, как тестировать асинхронный код, написанный с использованием Reactor 3 и любых других библиотек, совместимых со стандартом Reactive Streams. Рассмотрели основные особенности тестирования реактивных приложений Spring на основе модуля WebFlux с использованием модуля Spring Test. Познакомились с возможностью тестирования контроллера с использованием WebTestClient. Также мы узнали, как выполнить тестирование всей системы в целом, познакомились с несколькими советами по тестированию безопасности, которая также является важной частью современных веб-приложений. В заключение в этой главе рассмотрели несколько советов и рекомендаций по тестированию WebSocket. Здесь мы увидели некоторые ограничения модуля Spring Test 5.0.x, тем не менее узнали, как с помощью WebSocketClient создать тестируемый поток данных и проверить правильность взаимодействия «клиент – сервер». К сожалению, мы также обнаружили, что не существует простого способа симитировать с помощью WebSocketClient произвольных взаимодействий между серверами.

Итак, мы завершили тестирование нашей системы, и теперь пришло время узнать, как развернуть веб-приложение в облаке и как контролировать его работу в условиях нормальной эксплуатации. В следующей главе познакомимся с Pivotal Cloud – набором инструментов, помогающим контролировать всю реактивную систему. Мы также расскажем, как Spring 5 помогает решать проблемы.

Глава 10

И наконец, выпуск!

Мы рассмотрели все, что касается поддержки реактивности в Spring 5, включая идеи и приемы реактивного программирования с Reactor 3, новые функции Spring Boot 2, Spring WebFlux, Reactive Spring Data, Spring Cloud Streams и методы тестирования реактивных приложений. Пришло время подготовить наше реактивное приложение к эксплуатации. Оно должно предоставлять журналы, метрики, трассировки, переключение возможностей и другую информацию, обеспечивающую успешную работу. Приложение также должно обнаруживать зависимости времени выполнения, такие как база данных или брокер сообщений, не нарушая сообщений безопасности. Учитывая все это, можем создать выполняемый артефакт, готовый для развертывания в локальном или облачном окружении.

Будут рассмотрены следующие темы:

- проблемы эксплуатации программного обеспечения;
- потребность в показателях функционирования;
- цели и особенности Spring Boot Actuator;
- расширение возможностей Spring Boot Actuator;
- библиотеки и методы мониторинга реактивных приложений;
- трассировка взаимодействий между службами внутри реактивной системы;
- советы и рекомендации по развертыванию приложения в облаке.

Важность поддержки идеологии DevOps в приложениях

Практически любое программное обеспечение можно рассматривать с трех разных сторон. Каждая сторона представляет потребности своей целевой аудитории, занимающейся системой:

- пользователи заинтересованы в нормальной работе приложения и доступности всех необходимых особенностей;
- разработчики заинтересованы в удобстве разработки системы;

- сотрудники из подразделения эксплуатации больше заинтересованы в удобстве дальнейшего развития и эксплуатации (DevOps).

Теперь рассмотрим аспект эксплуатации программной системы. С точки зрения сотрудника подразделения эксплуатации, программная система должна поддерживать возможность простого развития и эксплуатации. Это означает, что система предлагает все необходимые проверки работоспособности и метрики, позволяет измерять свою производительность и бесшовно обновлять различные компоненты. Кроме того, поскольку в настоящее время все чаще по умолчанию используется архитектура микросервисов, система должна иметь средства мониторинга даже для развертывания в облаке. Без надлежащей инфраструктуры мониторинга наше программное обеспечение не сможет выжить в производственном окружении дольше нескольких дней.

Появление облачных систем для развертывания приложений упростило и демократизировало процессы доставки и эксплуатации программного обеспечения, предложив соответствующую инфраструктуру даже для самых требовательных программных разработок. IaaS, PaaS и системы управления контейнерами, такие как Kubernetes (<https://kubernetes.io>) или Apache Mesos (<http://mesos.apache.org>), устранили многие сложности, связанные с настройкой ОС и Сети, с резервным копированием файлов, сбором параметров, автоматическим масштабированием и многим-многом другим. Однако такие внешние службы и системы все еще не могут самостоятельно определить, обеспечивает ли наше приложение надлежащее качество обслуживания. Кроме того, поставщик облачных услуг не может знать, насколько эффективно используются основные ресурсы в зависимости от задач, которые выполняет наша система. Такая ответственность по-прежнему лежит на плечах разработчиков программного обеспечения и подразделений DevOps.

Для эффективной работы программного обеспечения, состоящего из десятков, сотен и иногда даже тысяч служб, нам нужны некоторые способы, позволяющие:

- идентифицировать службы;
- проверять работоспособность служб;
- определять параметры функционирования;
- исследовать журналы и динамически изменять уровень журналирования;
- отслеживать запросы и потоки данных.

Рассмотрим каждую задачу по очереди. Идентификация служб является обязательной в архитектуре микросервисов. Это связано с тем, что в большинстве случаев некоторые системы координации (например, Kubernetes) порождают множество экземпляров служб на разных узлах, перетасовывают их, создают и уничтожают узлы по мере увеличения и уменьшения потребностей клиентов. Несмотря на то что контейнеры или выполняемые файлы JAR обычно имеют значимые имена, важно иметь возможность идентифицировать имя службы, тип,

версию, время сборки и номер исправления во время выполнения. Это позволяет обнаружить неверную или ошибочную версию службы в производственной среде, отследить изменения, которые привели к регрессии (если имеются), и автоматически выяснить характеристики производительности разных версий одной и той же службы.

Кроме возможности различать службы во время выполнения нам также важно знать, все ли службы исправны. Если нет, то является ли отказавшая служба критически значимой. Конечные точки проверки работоспособности службы часто используются самой системой координации для идентификации и перезапуска аварийной службы. Здесь не обязательно иметь только два состояния: исправное и неисправное. Обычно состояние определяется как набор проверок – некоторые из них являются критическими, другие – нет. Например, степень работоспособности службы можно определять по размеру очереди, частоте появления ошибок, доступному дисковому пространству и свободной памяти. При определении общего состояния работоспособности стоит учитывать только основные показатели. В других случаях можно рискнуть создать службу, которую вряд ли можно считать работоспособной. В целом способность вернуть описание состояния в ответ на запрос означает, что служба, по крайней мере, способна обслуживать запросы. Эта характеристика часто используется системами управления контейнерами для проверки доступности службы и принятия решения о ее перезапуске.

Даже когда служба работает правильно и возвращает признак исправности, нам часто требуется более глубокое понимание деталей ее работы. Успешная система не только состоит из исправных компонентов, но и ведет себя предсказуемым и приемлемым образом. Список важнейших метрик системы может включать среднее время ответа, частоту появления ошибок и в зависимости от сложности запроса время обработки. Понимание того, как система ведет себя под нагрузкой, позволяет нам не только адекватно ее масштабировать, но и планировать расходы на инфраструктуру. Это также позволяет выявить «горячие» участки кода, неэффективные алгоритмы и ограничивающие факторы для масштабируемости. Оперативные метрики дают слепок текущего состояния системы и имеют большую ценность, но они гораздо информативнее, когда собираются непрерывно. Оперативные метрики помогают выявить тенденции и могут дать представление о некоторых взаимосвязанных характеристиках, таких как использование памяти сервера в соотношении со временем безотказной работы. Правильно организованный сбор метрик не требует много ресурсов сервера, но для сохранения истории метрик во времени требуется некоторая дополнительная инфраструктура, обычно база данных временных рядов, такая как Graphite (<https://graphiteapp.org>), InfluxDB (<https://www.influxdata.com>) или Prometheus (<https://prometheus.io>). Для визуализации временных рядов в дашбордах и настройки оповещений в ответ на критические ситуации часто требуется дополнительное программное обеспечение, такое как Grafana (<https://grafana.com>) или Zabbix (<https://www.zabbix.com>). Облачные платформы нередко предоставляют такое программное обеспечение своим клиентам в виде дополнительных услуг.

При мониторинге характеристик службы и расследовании инцидентов команда DevOps часто читает журналы. В идеале все журналы приложений должны храниться или, по крайней мере, анализироваться в централизованном месте. Для этой цели в экосистеме Java часто используется стек ELK (<https://www.elastic.co/elkstack>), включающий Elasticsearch, Logstash и Kibana. Хотя такой программный стек позволяет рассматривать десятки служб как одну систему, очень неэффективно передавать данные по Сети и хранить журналы для всех уровней журналирования. Обычно достаточно сохранить сообщения INFO и включить уровни DEBUG или TRACE, только чтобы исследовать некоторые повторяющиеся аномалии или ошибки. Для динамического управления уровнем журналов нам понадобятся некоторые бесппроблемные интерфейсы.

Когда журналов недостаточно, чтобы получить полную картину происходящего, последняя возможность избежать трудоемкой отладки – трассировка процессов внутри нашего программного обеспечения. Трассировка часто помогает получить подробную картину недавних запросов к серверу или описать полную топологию последующих запросов, включая время нахождения в очередях, длительность обработки запросов к базам данных, внешние вызовы с идентификаторами и т.д. Трассировка очень полезна для визуализации обработки запросов в режиме реального времени и незаменима при решении задачи улучшения производительности программного обеспечения. Распределенная трассировка делает то же самое в распределенной системе, позволяя отслеживать все запросы, сообщения, задержки в сети, ошибки и т.д. Далее расскажем, как организовать распределенную трассировку с помощью Spring Cloud Sleuth и Zipkin (<https://zipkin.io>).



Адриан Коул (Adrian Cole), руководитель проекта Zipkin, рассказал о важности различных перспектив в мониторинге приложений в следующей статье: <https://www.dotconferences.com/2017/04/adrian-cole-observability-3-ways-logging-metrics-tracing>.

Особенно важно то, что для успешной работы программной системы необходимо или, по крайней мере, желательно использовать все упомянутые методы. К счастью, в экосистеме Spring есть Spring Boot Actuator.

Все упомянутые выше методы оперативного контроля достаточно просты и хорошо описаны на примере обычного приложения на основе сервлетов. Однако, поскольку реактивное программирование на платформе Java все еще является новинкой, для достижения аналогичных целей в реактивных системах могут потребоваться некоторые модификации кода или даже придется использовать совершенно другие подходы к реализации.

В любом случае, с точки зрения эксплуатации служба, реализованная с применением методов реактивного программирования, не должна сильно отличаться от обычной синхронной службы. Следования рекомендациям руководства «Двенадцать факторов» (<https://12factor.net>) должно быть достаточно для создания программных продуктов, простых в обслуживании и развитии.

Мониторинг реактивных Spring-приложений

Вообще говоря, всю инфраструктуру мониторинга Spring-приложения можно реализовать вручную, но это слишком большая трата сил и времени, особенно если делать это снова и снова для каждого микросервиса в системе. К счастью, Spring Framework предлагает очень неплохой набор инструментов для создания приложений, дружественных к методологии DevOps. Называется этот набор инструментов **Spring Boot Actuator**. Добавив лишь одну дополнительную зависимость Spring Boot, вы получаете некоторые ценные возможности, а также каркас инфраструктуры мониторинга.

Spring Boot Actuator

Spring Boot Actuator – это подпроект Spring Boot, который предлагает множество готовых к использованию функций для применения в приложениях Spring Boot. В это множество входят функции, возвращающие служебную информацию, осуществляющие проверку работоспособности, выполняющие сбор метрик, отслеживающие трафик, оценивающие состояние базы данных и т.д. Основная идея Spring Boot Actuator заключается в предоставлении комплекса основных метрик приложения и простой возможности его расширения.

Spring Boot Actuator реализует конечные точки HTTP и компоненты JMX с большим количеством оперативной информации и обеспечивает бесшовную интеграцию с широким спектром систем мониторинга. Его многочисленные плагины еще больше расширяют эти возможности. Как и в большинстве модулей Spring, достаточно добавить одну зависимость, чтобы получить согласованный набор инструментов для мониторинга приложений.

Добавление механизма мониторинга в проект

Чтобы добавить механизм мониторинга в проект, достаточно включить следующую зависимость в файл сборки Gradle:

```
compile('org.springframework.boot:spring-boot-starter-actuator')
```

Здесь мы полагаемся на тот факт, что фактическая версия библиотеки и все ее зависимости определены в спецификации **Spring Boot Bill of Materials (BOM)**.

Механизм Spring Boot Actuator первоначально появился в версии Spring Boot 1.x, но в Spring Boot 2.x его значительно улучшили. На данный момент этот механизм является технологически независимым и поддерживает как Spring Web MVC, так и Spring WebFlux. Он также использует одну и ту же модель безопасности, что

и остальная часть приложения, поэтому его можно внедрить в реактивное приложение и использовать все преимущества реактивного программирования.

Конфигурация механизма мониторинга по умолчанию предоставляет все конечные точки в URL `/actuator`, но его можно изменить, переопределив свойство `management.endpoints.web.basepath`. То есть после запуска приложения можно сразу начать исследовать работу службы.



Поскольку Spring Boot Actuator в значительной степени зависит от веб-инфраструктуры приложения, в качестве зависимости к приложению следует подключить Spring Web MVC или Spring WebFlux.

Теперь рассмотрим основные задачи, перечисленные выше, и обсудим способы их реализации в реактивном приложении.

Конечная точка для получения информации о службе

По умолчанию Spring Boot Actuator предоставляет наиболее ценную информацию для мониторинга, масштабирования и развития системы с течением времени. В зависимости от конфигурации приложения он возвращает информацию об исполняемом артефакте приложения (группа служб, идентификатор артефакта, имя артефакта, версия артефакта, время сборки) и координаты Git (имя ветви, идентификатор фиксации, время фиксации). Конечно, мы можем расширить список дополнительной информацией, если необходимо.



Чтобы вернуть информацию, собранную во время сборки приложения, можем добавить следующую конфигурацию в файл сборки Gradle в разделе `springBoot: buildInfo()`. Конечно, такая же функциональность доступна для сборок Maven. Более подробную информацию можно найти в статье <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build>.

Actuator отображает информацию о приложении в конечной точке `/actuator/info`. Обычно эта конечная точка включена по умолчанию, но ее доступность можно настроить свойством `management.endpoint.info.enabled`.

Мы можем добавить свои сведения в информацию, возвращаемую этой конечной точкой REST, добавив в файл `application.property` следующие строки.

```
info:
  name: "Reactive Spring App"
  mode: "testing"
  service-version: "2.0"
  features:
    feature-a: "enabled"
    feature-b: "disabled"
```

С другой стороны, ту же информацию можно получить программно, зарегистрировав компонент, реализующий интерфейс `InfoContributor`, как показано ниже.

```
@Component
public class AppModeInfoProvider implements InfoContributor { // (1)
    private final Random rnd = new Random();

    @Override
    public void contribute(Info.Builder builder) { // (2)
        boolean appMode = rnd.nextBoolean();
        builder
            .withDetail("application-mode", // (3)
                appMode ? "experimental" : "stable");
    }
}
```

Здесь мы реализовали интерфейс `InfoContributor` (1) с единственным методом-построителем `contrib(...)` (2), который можно использовать для добавления необходимой информации с помощью `withDetail(...)` (3), как показано на рис. 10.1.

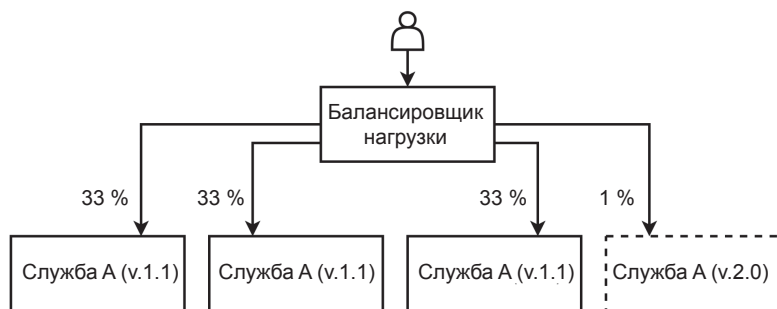


Рис. 10.1. Пример реализации шаблона канареечного развертывания (Canary Deployment). Балансировщик нагрузки направляет трафик экземплярам службы в соответствии с их версиями

При обсуждении расширенных сценариев конечную точку, возвращающую информацию о службе, можно использовать при выполнении канареечного развертывания (<https://martinfowler.com/bliki/CanaryRelease.html>). Здесь балансировщик нагрузки может использовать информацию о версии службы для маршрутизации входящего трафика.



Подробности о конечной точке `/info` Spring Boot Actuator можно найти по ссылке: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info>.

Поскольку Spring Boot Actuator не предоставляет никаких реактивных или асинхронных API для передачи информации, реактивная служба ничем не отличается от блокирующей.

Конечная точка для получения информации о работоспособности

Следующим важным пунктом мониторинга системы является возможность проверки работоспособности службы. В наиболее простом случае работоспособность службы можно интерпретировать как способность отвечать на запросы. Такой сценарий изображен на рис. 10.2.

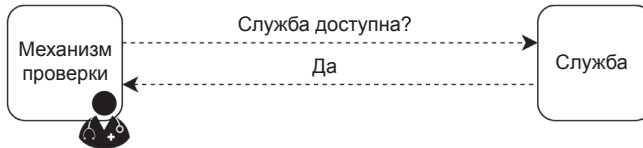


Рис. 10.2. Пример упрощенной проверки работоспособности службы путем проверки ее доступности

Основная проблема заключается в том, что служба может быть доступна через Сеть. Однако некоторые жизненно важные компоненты, такие как жесткий диск (если используется), база данных или зависящая служба, могут быть недоступны. Из-за этого служба может оказаться не полностью функциональной. С точки зрения эксплуатации проверка работоспособности означает гораздо больше, чем простая проверка доступности. Прежде всего работоспособная служба – это служба, в которой также доступны все ее компоненты. Кроме того, информация о работоспособности должна включать все детали, которые позволят оперативному персоналу реагировать на потенциальные угрозы отказа как можно скорее. Обнаружив недоступность базы данных или нехватку свободного места на диске, оперативный персонал сможет предпринять соответствующие действия. Таким образом, информация о работоспособности может включать больше деталей, как показано на рис. 10.3.

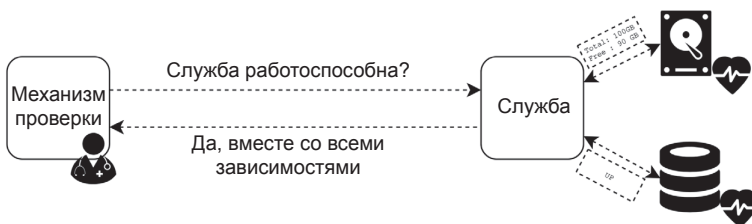


Рис. 10.3. Пример проверки работоспособности службы с предоставлением дополнительных сведений об используемых ресурсах

К счастью, Spring Boot Actuator предлагает полноценный подход к мониторингу работоспособности. К важным деталям, касающимся работоспособности службы, можно получить доступ через конечную точку `/actuator/health`. Эта конечная точка включена по умолчанию. Кроме того, Spring Boot Actuator поддерживает обширный список встроенных индикаторов работоспособности для наиболее распространенных компонентов, таких как Cassandra, MongoDB, JMS и другие популярные службы, интегрированные в экосистему Spring.



Подробности о встроенных индикаторах HealthIndicator можно найти по ссылке <https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#autoconfiguredhealthindicators>.

Наряду со встроенными индикаторами мы можем предоставить свою реализацию HealthIndicator. Однако наиболее интересной частью Spring Boot Actuator 2.0 является полная интеграция с Spring WebFlux и Reactor 3, что позволяет использовать в индикаторах новый реактивный интерфейс, который называется ReactiveHealthIndicator. Это может иметь жизненно важное значение, когда для оценки работоспособности требуются дополнительные запросы, которые можно обрабатывать более эффективно с помощью WebClient, как описано в главе 6 «Неблокирующие и асинхронные взаимодействия с WebFlux». Следующий пример демонстрирует реализацию нестандартного индикатора с использованием нового API.

```
@Component
class TemperatureSensor {                                     // (1)
    public Mono<Integer> batteryLevel() {                     // (1.1)
        // Здесь выполняется сетевой запрос
    }
    ...
}

@Component
class BatteryHealthIndicator implements ReactiveHealthIndicator { // (2)
    private final TemperatureSensor temperatureSensor;        // (2.1)

    @Override
    public Mono<Health> health() {                             // (3)
        return temperatureSensor
            .batteryLevel()
            .map(level -> {
                if (level > 40) {
                    return new Health.Builder()
                        .up()                                   // (4)
                        .withDetail("level", level)
                        .build();
                } else {
                    return new Health.Builder()
                        .status(new Status("Low Battery"))      // (5)
                        .withDetail("level", level)
                        .build();
                }
            })
            .onErrorResume(err -> Mono.just(new Health.Builder()
                .outOfService()                                // (6.1)
            ))
    }
}
```

```
        .withDetail("error", err.getMessage())
        .build()
    );
}
```

Этот пример показывает, как организовать взаимодействие с внешним датчиком, установленным в доме и доступным по Сети.

1. Служба `TemperatureSensor` имеет метод `Mono<Integer> batteryLevel()` (1.1), который посылает запрос датчику и, если он доступен, возвращает текущий уровень заряда аккумулятора по шкале от 0 до 100 %. Этот метод возвращает `Mono` с ответом и использует `WebClient` для эффективного взаимодействия.
2. Чтобы использовать данные об уровне заряда аккумулятора при определении работоспособности службы, определяем собственный класс `BatteryHealthIndicator`. Он реализует интерфейс `ReactiveHealthIndicator` и имеет ссылку на службу `TemperatureSensor` (2.1).
3. Для реализации интерфейса индикатор работоспособности определяет метод `Mono<Health> health()`, возвращающий реактивный тип. Следовательно, состояние работоспособности можно получать реактивно, когда от датчика температуры поступает ответ.
4. В зависимости от заряда аккумулятора возвращаем предопределенный код UP с дополнительной информацией.
5. Пример нестандартной реализации `Health`. В этом примере получаем состояние `Low Battery` (низкий заряд).
6. Используя возможности `Reactor`, можно реагировать на ошибки связи и возвращать статус `OUTOFSERVICE` с некоторыми подробностями о фактической ошибке (6.1).



`Spring Boot Actuator` имеет несколько режимов, которые определяют, когда поставлять более подробную информацию о работоспособности, а когда показывать только высокоуровневое состояние (UP, DOWN, OUTOFSERVICE, UNKNOWN). Для тестирования достаточно присвоить значение `always` (всегда) свойству `application.endpoint.health.show-details`. Более подробно доступные параметры описаны в статье <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-health>. Кроме того, чтобы не перегружать датчик и не ограничивать скорость работы, можно кешировать уровень заряда батареи на некоторый период времени, используя возможности `Reactor`, или настроить кеширование с помощью свойства `management.endpoint.health.cache.time-to-live = 10s`.

Подобный подход к мониторингу позволяет организовывать соответствующие действия, такие как создание запроса на замену батареи или отправка уведомления оперативному персоналу о низком заряде батареи.



Spring Boot Actuator 2 предлагает несколько встроенных Reactive HealthIndicator, информацию о которых можно найти по ссылке <https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#autoconfiguredreactivehealthindicators>.

Конечная точка для получения информации о параметрах работы

Следующее важное условие успешного мониторинга приложения – сбор оперативных параметров работы. Конечно, Spring Boot Actuator охватывает и этот аспект и обеспечивает мониторинг основных параметров работы JVM, таких как время безотказной работы, использование памяти, загрузка процессора и паузы на сборку мусора. WebFlux также предлагает некоторую статистику, касающуюся обработки входящих HTTP-запросов. Однако, чтобы обеспечить осмысленное понимание происходящего в службе с точки зрения бизнеса, мы должны самостоятельно расширить набор оперативных параметров.

Начиная с версии Spring Boot 2.0 Actuator изменил базовую библиотеку, используемую для сбора параметров, и теперь использует библиотеку Micrometer (<https://micrometer.io>).

Если конечная точка `/actuator/metrics` включена, она возвращает список отслеживаемых параметров и позволяет перейти к нужному датчику или таймеру, а также предоставляет дополнительную контекстную информацию в виде тегов. Информация об одном из параметров, скажем, `jvm.gc.pause`, может выглядеть следующим образом:

```
{
  "name": "jvm.gc.pause",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 5
    },
    {
      "statistic": "TOTALTIME",
      "value": 0.347
    }
  ],
  "availableTags": [
    {
      "tag": "cause",
      "values": [
```

```

        "Heap Dump Initiated GC",
        "Metadata GC Threshold",
        "Allocation Failure"
    ]
}
]
}

```

Одним из недостатков новой конечной точки `/actuator/metrics` по сравнению с аналогом Spring Boot 1.x является невозможность извлечения всех отслеживаемых параметров одним запросом REST. Впрочем, это незначительная проблема.

Как обычно, мы можем зарегистрировать новые параметры, используя реестр параметров Micrometer. Далее рассмотрим механику этого процесса и посмотрим, какие оперативные параметры могут пригодиться для мониторинга реактивного приложения.



Конечная точка `/actuator/metrics` выключена по умолчанию. Конфигурация по умолчанию предоставляет только конечные точки `info` и `health`. Поэтому конечную точку `/actuator/metrics` нужно включить, определив свойство `management.endpoints.web.exposure.include: info, health, metrics` в файле `application.property`. То же относится ко всем конечным точкам, которые упоминаются далее. Если необходимо включить все конечные точки, можно использовать прием с шаблонным символом: `management.endpoints.web.exposure.include: *`. За дополнительной информацией обращайтесь по ссылке <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints-enabling-endpoints>.

Конечная точка управления журналированием

Spring Boot Actuator предлагает также две конечные точки для управления журналированием. Первая – `/actuator/loggers` – позволяет получить доступ к журналу и изменять уровень журналирования во время выполнения без перезапуска приложения. Это очень полезная конечная точка, поскольку переключение уровня детализации журналирования без перезапуска службы является важной составляющей успешной работы приложения. Имея богатый опыт отладки реактивных приложений, мы можем ответственно утверждать, что возможность переключения уровня журналирования и анализа результатов на лету имеет решающее значение. Конечно, не очень удобно изменять уровни с консоли с помощью команды `curl`, но эта функция хорошо работает с инструментом администрирования Spring Boot Admin, который предлагает удобный пользовательский интерфейс для таких целей.

Эта конечная точка позволяет включать или отключать динамическое журналирование с помощью оператора Reactor `log()`, который был описан в главе 4

«Project Reactor – основа реактивных приложений». Следующий пример изображает реактивную службу, которая отправляет события в поток SSE.

```
@GetMapping(path = "/temperature-stream",
    produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<TemperatureDto> temperatureEvents() {
    return temperatureSensor.temperatureStream()
        .log("sse.temperature", Level.FINE)           // (1)
        .map(this::toDto);
}
```

Здесь оператор `log()` регистрирует температуру `sse.tagger` с уровнем `Level.FINE` (1). Мы можем динамически включать или отключать вывод этого оператора, используя конечную точку `/actuator/loggers`.

С другой стороны, иногда может пригодиться возможность доступа к журналам приложений без копирования файлов с удаленного сервера. С этой целью Spring Boot Actuator предоставляет конечную точку `/actuator/logfile`, которая возвращает файл с журналами по Сети. Интерфейс администратора Spring Boot Admin также имеет удобную веб-страницу, которая отображает потоки журналов приложения в пользовательском интерфейсе.



Недоступность конечной точки `/actuator/logfile` может быть связана с отсутствием настроек файлов журналов в приложении. В этом случае следует добавить параметр `logging.file: my.log`.

Другие важные конечные точки

Наряду с упомянутыми ранее конечными точками Spring Boot Actuator предлагает целый ряд удобных конечных точек. Соответствующая документация доступна по ссылке: <https://docs.spring.io/spring-boot/docs/current/actuator-api/html>.

Ниже перечислены и кратко описаны наиболее важные из них (все конечные точки доступны относительно базового URL `/actuator`):

- `/configprops`: предоставляет доступ ко всем конфигурационным свойствам в приложении;
- `/env`: предоставляет доступ к переменным окружения;
- `/mappings`: предоставляет доступ ко всем конечным точкам приложения;
- `/httptrace`: предоставляет доступ к трассировочной информации о взаимодействиях через HTTP на стороне сервера и клиента;
- `/auditevents`: предоставляет доступ ко всем контролируемым событиям в приложении;
- `/beans`: предоставляет доступ к списку компонентов, доступных в контексте Spring;

- /caches: предоставляет доступ к средствам управления кешированием в приложении;
- /sessions: возвращает список активных HTTP-сеансов;
- /threaddump: позволяет получить дамп потока выполнения приложения в JVM;
- /heapdump: позволяет сгенерировать и загрузить дамп кучи. Полученный в результате файл имеет формат HPROF: <https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html>.

Реализация своей конечной точки для Actuator

Spring Boot Actuator позволяет регистрировать нестандартные конечные точки, которые не только отображают данные, но и могут управлять поведением приложения. Для этого компонент Spring должен быть снабжен аннотацией `@Endpoint`. Для регистрации операций чтения/записи в качестве конечной точки Actuator библиотека предоставляет аннотации `@ReadOperation`, `@WriteOperation` и `@DeleteOperation`, которые, в свою очередь, отображаются в HTTP-методы GET, POST и DELETE соответственно. Обратите внимание, что сгенерированные конечные точки REST потребляют и генерируют контент следующего типа: `application/vnd.spring-boot.actuator.v2+json`, `application/json`.

Чтобы показать, как эту возможность использовать в реактивном приложении, создадим свою конечную точку, которая информирует о текущем времени на сервере и о разнице во времени с **сервером времени NTP**. Иногда это помогает решить проблемы, связанные с неправильным временем из-за неверной конфигурации сервера или Сети. Это особенно удобно для распределенных систем. Для этого зарегистрируем компонент с аннотацией `@Endpoint` и настроенным идентификатором (в нашем случае `server-time`). Также вернем интересные нас данные через метод с аннотацией `@ReadOperation`.

```
@Component
@Endpoint(id = "server-time") // (1)
public class ServerTimeEndpoint {
    private Mono<Long> getNtpTimeOffset() { // (2)
        // Фактический сетевой вызов для получения
        // смещения текущего времени
    }

    @ReadOperation // (3)
    public Mono<Map<String, Object>> reportServerTime() { // (4)
        return getNtpTimeOffset() // (5)
            .map(timeOffset -> { // (6)
                Map<String, Object> rsp = new LinkedHashMap<>(); //
                rsp.put("serverTime", Instant.now().toString()); //
            })
    }
}
```

```
        rsp.put("ntpOffsetMillis", timeOffset);           //  
        return rsp;  
    });  
}
```

Предыдущий пример показывает, как создать свою конечную точку `@Endpoint` с использованием реактивных типов из Project Reactor. Вот некоторые пояснения к примеру.

1. Класс конечной точки с аннотацией `@Endpoint`. Здесь аннотацией `@Endpoint` отмечен класс `ServerTimeEndpoint`, который будет играть роль конечной точки REST. Параметр `id` аннотации `@Endpoint` определяет URL конечной точки. В такой комбинации URL будет иметь вид `/actuator/server-time`.
2. Определение метода, который асинхронно возвращает смещение относительно текущего времени на сервере NTP. Метод `getNtpTimeOffset()` возвращает `Mono<Long>`, поэтому он может использоваться реактивным способом.
3. Аннотация `@ReadOperation` отмечает метод `reportServerTime()`, который должен быть доступен через механизмы Actuator как REST GET операция.
4. Метод `reportServerTime()`, возвращает реактивный тип `Mono<Map<String, Object>>`, поэтому его эффективно можно использовать в приложении Spring WebFlux.
5. Объявление асинхронного вызова сервера NTP, результат которого определяет начало реактивного потока данных.
6. Операция преобразования результата. Когда приходит ответ, он преобразуется оператором `map()` в ассоциативный массив с текущим временем на сервере и смещением относительно NTP, который затем возвращается пользователю. Обратите внимание, что здесь также можно использовать все методы обработки ошибок, описанные в главе 4 «Project Reactor – основа реактивных приложений».



ТИП NTP (<http://www.ntp.org>) – популярный протокол для синхронизации часов компьютера по Сети. Серверы NTP общедоступны, но будьте осторожны и избегайте отправки слишком большого числа запросов, потому что NTP-серверы могут заблокировать IP-адрес вашего приложения.

С помощью своих конечных точек можно быстро реализовать переключение динамических функций, отслеживание пользовательских запросов, управление режимом работы (ведущий/ведомый), удаление старых сеансов и множество других функций, которые имеют отношение к работе приложения. Теперь все это можно реализовать с использованием приемов реактивного программирования.

Безопасность конечных точек

В предыдущих разделах мы рассмотрели основы Spring Boot Actuator и реализовали свою конечную точку. Обратите внимание, что Spring Boot Actuator может

по неосторожности раскрыть не только общедоступную, но и конфиденциальную информацию. Доступ к переменным среды, к структуре приложения, свойствам конфигурации, способности делать дампы кучи и потоков выполнения и другие факторы могут упростить жизнь злоумышленникам, пытающимся взломать приложение. В некоторых случаях Spring Boot Actuator может также предоставлять данные частных пользователей. Поэтому мы должны заботиться о безопасном доступе ко всем конечным точкам механизма мониторинга, так же как к обычным конечным точкам REST.

В связи с тем что Spring Boot Actuator использует ту же модель безопасности, что и остальная часть приложения, разрешения можно легко настроить там же, где определена основная конфигурация безопасности. Например, следующий код разрешает доступ к конечной точке `/actuator/` только пользователям, имеющим полномочия `ACTUATOR`.

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(
    ServerHttpSecurity http
) {
    return http.authorizeExchange()
        .pathMatchers("/actuator/").hasRole("ACTUATOR")
        .anyExchange().authenticated()
        .and().build();
}
```

Конечно, политики доступа можно настроить по-разному. Обычно неаутентифицированный доступ можно предоставлять к конечным точкам `/actuator/info` и `/actuator/health`, которые часто используются для идентификации приложений и процедур проверки работоспособности, и закрывать доступ к другим конечным точкам, которые потенциально могут возвращать конфиденциальную информацию или информацию, которую можно использовать для организации атаки на систему.

С другой стороны, можно организовать доступ ко всем конечным точкам с функциями управления на отдельном порту и настроить правила доступа к Сети, чтобы все управление осуществлялось только через внутреннюю виртуальную сеть. Чтобы обеспечить такую конфигурацию, достаточно присвоить свойству `management.server.port` требуемый порт HTTP.

В целом Spring Boot Actuator предлагает множество функций, упрощающих идентификацию, мониторинг и управление приложениями. Благодаря тесной интеграции с Spring WebFlux Actuator 2.x эффективно использует ресурсы, обеспечивая поддержку большинства его реактивных конечных точек. Actuator 2.x упрощает общий процесс разработки, не доставляя особых хлопот, расширяя возможности приложений, консолидируя лучшее поведение по умолчанию и делая жизнь команды DevOps проще.

Механизм Spring Boot Actuator хорош уже сам по себе, но он еще более полезен в сочетании с инструментами автоматического сбора информации и предоставления ее в наглядной форме – в виде диаграмм, графиков и предупреждений. Поэтому далее мы рассмотрим интересный модуль под названием Spring Boot Admin, который позволяет получить доступ ко всей важной управляющей информации из нескольких служб с помощью единого удобного пользовательского интерфейса.

Micrometer

Начиная с версии Spring Boot 2.0 изменилась библиотека, используемая в Spring Framework по умолчанию для сбора параметров. Ранее использовалась библиотека Dropwizard Metrics (<https://metrics.dropwizard.io>), а теперь совершенно новая библиотека под названием Micrometer (<https://micrometer.io>). Micrometer – это автономная библиотека с минимальным списком зависимостей. Она разрабатывается как отдельный проект, но нацелена на Spring Framework как основного потребителя. Библиотека предоставляет фасад для клиентов самых популярных систем мониторинга. Micrometer предлагает независимый от производителя API мониторинга, подобно тому как SLF4J предлагает API для журналирования. На данный момент Micrometer хорошо интегрируется с Prometheus, Influx, Netflix Atlas и десятком других систем мониторинга. Она также имеет встроенное хранилище параметров в памяти, которое позволяет использовать библиотеку даже без внешней системы мониторинга.

Библиотека Micrometer предназначена для поддержки многомерных параметров, когда каждый параметр, кроме имени, содержит теги в виде пары ключ/значение. Такой подход позволяет просматривать агрегированные значения, а также анализировать содержимое тегов при необходимости. Когда целевая система мониторинга не поддерживает многомерные параметры, библиотека разворачивает связанные теги и добавляет их к имени.



Узнать больше об устройстве библиотеки Micrometer, о ее API и поддерживаемых системах мониторинга можно на сайте проекта <https://micrometer.io>.

Spring Boot Actuator использует библиотеку Micrometer для вывода параметров приложения через конечную точку `/actuator/metrics`. С модулем Actuator мы получаем компонент `MeterRegistry`, автоматически получающий настройки по умолчанию. Это интерфейс Micrometer, который защищает код клиента от информации о том, как и где хранятся все параметры. Для каждой поддерживаемой системы мониторинга имеется своя реализация интерфейса `MeterRegistry`, поэтому, например, для поддержки Prometheus приложение должно создавать экземпляр класса `PrometheusMeterRegistry`. Также Micrometer предлагает `CompositeMeterRegistry` для вывода показателей сразу в несколько систем. Кроме того, интерфейс `MeterRegistry` является точкой входа, которая позволяет добавлять свои параметры из пользовательского кода.

Существуют разные типы параметров (такие как `Timer`, `Counter`, `Gauge`, `DistributionSummary`, `LongTaskTimer`, `TimeGauge`, `FunctionCounter`, `FunctionTimer`), и все они расширяют интерфейс `Meter`. Каждый тип параметров имеет универсальный API и позволяет настраивать поведение мониторинга желаемым образом.

Параметры по умолчанию в Spring Boot

По умолчанию Spring Boot Actuator настраивает `Micrometer` для сбора наиболее часто используемых параметров, таких как время работы процесса, загрузка процессора, использование памяти, паузы на сборку мусора, количество шагов, загруженные классы и количество дескрипторов открытых файлов. Он также подсчитывает события `Logback`. Все это поведение определяется в `MetricsAutoConfiguration` и дает довольно очевидную картину в отношении параметров работы.

Для реактивных приложений Actuator поддерживает экземпляр `WebFluxMetricsAutoConfiguration`, который добавляет специализированный `WebFilter`. Этот фильтр добавляет обратные вызовы в `ServerWebExchange`, чтобы определить момент завершения обработки запроса. Таким образом можно определить время обслуживания запроса. Кроме того, данный фильтр включает в массив параметров такую информацию, как URI запроса, метод HTTP, состояние ответа и тип исключения (если есть) в форме тегов. Таким образом, реактивную обработку HTTP-запроса можно легко измерить с помощью библиотеки `Micrometer`. Результаты возвращаются с параметром `http.server.requests`.

Аналогично Spring Boot Actuator использует `RestTemplate` и регистрирует параметр `http.client.requests` для получения информации об исходящих запросах. Начиная с версии Spring Boot 2.1 Actuator делает то же самое с компонентом `WebClient.Builder`. Тем не менее даже в Spring Boot 2.0 легко можно добавить желаемые параметры рабочих процессов, связанных с `WebClient`. Такой подход объясняется в следующем разделе.

Кроме того, Spring Boot Actuator позволяет добавлять общие теги во все счетчики `Meter` приложения. Это особенно удобно при развертывании служб на нескольких узлах, поскольку позволяет четко различать узлы по тегам. Для этого приложение должно зарегистрировать компонент, реализующий интерфейс `MeterRegistryCustomizer`.



Узнать больше о настройках, которые Spring Boot Actuator выполняет в библиотеке `Micrometer`, можно в статье <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics-getting-started>.

Мониторинг реактивных потоков данных

В версии 3.2 в библиотеку Project Reactor была добавлена интеграция с библиотекой Micrometer. Project Reactor отлично работает и без нее, но, обнаружив присутствие Micrometer в пути к классам приложения, она может сообщать о некоторых элементарных параметрах, описывающих условия работы приложения. Теперь пришло время рассказать, как контролировать реактивные потоки, используя встроенные функции, и как добавить пользовательские счетчики для нужных параметров.



Project Reactor 3.2 входит в состав Spring Framework 5.1 и Spring Boot 2.1. Однако есть возможность использовать эту версию даже в приложениях на основе Spring Framework 5.0 и Spring Boot 2.0. Кроме того, весь код, связанный с регистрацией пользовательских параметров, должен нормально работать с предыдущими версиями Project Reactor.

Мониторинг потоков в Reactor

В Project Reactor 3.2 реактивные типы Flux и Mono получили оператор `metrics()`. Он сообщает об оперативных параметрах, имеющих отношение к текущему потоку данных. Оператор `metrics()` действует подобно оператору `log()`. Он взаимодействует с оператором `name()` для создания имени целевого параметра и добавления тега. Например, следующий код демонстрирует, как добавить параметры в исходящий поток SSE.

```
@GetMapping(
    path = "/temperature-stream",
    produces = MediaType.TEXT_EVENT_STREAM_VALUE)

public Flux<Temperature> events() {
    return temperatureSensor.temperatureStream()           // (1)
        .name("temperature.stream")                        // (2)
        .metrics();                                         // (3)
}
```

Здесь `temperatureSensor.temperatureStream()` возвращает `Flux<Temperature>` (1), в то время как метод `name("temperature.stream")` добавляет имя для точки мониторинга (2). Метод `metrics()` (3) регистрирует новую метрику в экземпляре `MeterRegistry`.

В результате библиотека Reactor регистрирует счетчики `reactor.subscribed` и `reactor.requested` и таймеры `reactor.flow.duration` и `reactor.onNext.delay`, каждый из которых имеет тег `flow` со значением `temperature.stream`. Эти параметры позволят нам отследить количество экземпляров потока, число запрашиваемых элементов, максимальное и общее время существования потока, а также задержку `onNext`.



Имейте в виду, что Reactor использует имя потока в теге `flow` в терминах параметров Micrometer. Поскольку библиотека Micrometer ограничивает количество отслеживаемых тегов, важно настроить предел для сбора информации обо всех интересующих потоках.

Мониторинг планировщиков в Reactor

Реактивные потоки обычно обслуживаются разными планировщиками, поэтому иногда полезно отслеживать более детализированные параметры, касающиеся их работы. В некоторых случаях можно использовать пользовательский `ScheduledThreadPoolExecutor` с параметрами, настроенными вручную. Например, предположим, что у нас есть следующий класс:

```
public class MeteredScheduledThreadPoolExecutor
    extends ScheduledThreadPoolExecutor {                                // (1)

    public MeteredScheduledThreadPoolExecutorShort(
        int corePoolSize,
        MeterRegistry registry                                        // (2)
    ) {
        super(corePoolSize);
        registry.gauge("pool.core.size", this.getCorePoolSize()); // (3)
        registry.gauge("pool.active.tasks", this.getActiveCount()); //
        registry.gauge("pool.queue.size", this.getQueue().size()); //
    }
}
```

Здесь класс `MeteredScheduledThreadPoolExecutor` расширяет обычный `ScheduledThreadPoolExecutor` (1) и дополнительно получает экземпляр `MeterRegistry` (2). В конструкторе мы регистрируем несколько датчиков (3), чтобы отслеживать размер пула потоков, количество активных заданий и размер очереди.

С небольшими модификациями такие исполнители могут также отслеживать количество успешных заданий и заданий, потерпевших неудачу, а также время, потраченное на их выполнение. Для таких целей реализация исполнителя должна также переопределять методы `beforeExecute()` и `afterExecute()`.

Такой настроенный исполнитель может использоваться в реактивном потоке, как показано ниже.

```
MeterRegistry meterRegistry = this.getRegistry();

ScheduledExecutorService executor =
    new MeteredScheduledThreadPoolExecutor(3, meterRegistry);

Scheduler eventsScheduler = Schedulers.fromExecutor(executor);
Mono.fromCallable(this::businessOperation)
```



```
.subscribeOn(eventsScheduler)
....
```

Несмотря на широкие возможности, подобный подход не распространяется на встроенные планировщики Scheduler, такие как `parallel()` или `elastic()`. Чтобы настроить все планировщики, поддерживаемые библиотекой Reactor, можно реализовать свою версию `Schedulers.Factory`. Например, взгляните на следующую реализацию такой фабрики:

```
class MetersSchedulersFactory implements Schedulers.Factory { // (1)
    private final MeterRegistry registry;

    public ScheduledExecutorService decorateExecutorService( // (2)
        String type, // (2.1)
        Supplier<? extends ScheduledExecutorService> actual // (2.2)
    ) {
        ScheduledExecutorService actualScheduler = actual.get(); // (3)
        String metric = "scheduler." + type + ".execution"; // (4)

        ScheduledExecutorService scheduledExecutorService =
            new ScheduledExecutorService() { // (5)
                public void execute(Runnable command) { // (6)
                    registry.counter(metric, "tag", "execute") // (6.1)
                        .increment();
                    actualScheduler.execute(command); // (6.2)
                }

                public <T> Future<T> submit(Callable<T> task) { // (7)
                    registry.counter(metric, "tag", "submit") // (7.1)
                        .increment();
                    return actualScheduler.submit(task); // (7.2)
                }

                // другие переопределенные методы ...
            };
        registry.counter("scheduler." + type + ".instances") // (8)
            .increment();
        return scheduledExecutorService; // (9)
    }
}
```

Пояснения к предыдущему примеру.

1. Собственная реализация класса `Schedulers.Factory`.
2. Объявление метода, который позволяет декорировать запланированную службу исполнителя, где `type` (2.1) представляет тип планировщика (`parallel`, `elastic`), а `actual` (2.2) содержит ссылку на фактическую службу, которая должна быть декорирована.

3. Извлекается фактический экземпляр службы исполнителя, определяемый декоратором.
4. Определение имени счетчика, включающего тип планировщика Scheduler для получения удобочитаемого имени параметра.
5. Объявление анонимного экземпляра ScheduledExecutorService, который декорирует фактический экземпляр ScheduledExecutorService и предлагает дополнительные возможности.
6. Переопределение метода `execute(Runnable command)`, который выполняет `MeterRegistry#counter` с дополнительным тегом `execute` (6.1) и увеличивает число полученных параметров на 1. Далее делегат вызывает фактическую реализацию метода (6.2).
7. По аналогии с методом `execute` мы переопределяем метод `Future<T> submit(Callable<T> task)`. Каждый раз, когда выполняется этот метод, он вызывает `MeterRegistry#counter` с дополнительным тегом `submit` и в конце увеличивает число вызовов в реестре (7.1). По окончании делегируем выполнение метода фактической службе (7.2).
8. Регистрация счетчика созданных экземпляров службы исполнителя.
9. Здесь мы наконец возвращаем декорированный экземпляр Scheduled ExecutorService с дополнительными параметрами Micrometer.

Чтобы использовать `MetersSchedulersFactory`, фабрику нужно зарегистрировать:

```
Schedulers.setFactory(new MeteredSchedulersFactory(meterRegistry));
```

Теперь все планировщики из Reactor (включая пользовательские, определенные в `MeteredScheduledThreadPoolExecutor`) можно будет оценить с помощью параметров Micrometer. Такой подход обеспечивает большую гибкость, но требует некоторой дисциплины, чтобы сбор параметров не превратился в самую ресурсоемкую часть приложения. Например, даже если поиск параметра (счетчика/таймера/датчика) в реестре будет выполняться очень быстро, полученную ссылку лучше сохранить в локальной переменной или в поле экземпляра, чтобы не производить избыточный поиск. Кроме того, не все аспекты работы приложения заслуживают отдельного набора метрик. Всегда желательно руководствоваться здравым смыслом и стараться не переборщить с определением параметров для мониторинга.

Реализация своих параметров Micrometer

В реактивные потоки легко можно добавить свою логику мониторинга, даже без встроенной поддержки. Например, следующий код показывает, как добавить счетчик вызовов для `WebClient`, который по умолчанию не поддерживает никаких параметров:

```
WebClient.create(serviceUri) // (1)
    .get()
    .exchange()
    .flatMap(cr -> cr.toEntity(User.class)) // (2)
    .doOnTerminate(() -> registry
        .counter("user.request", "uri", serviceUri) // (3)
        .increment())
```

Здесь мы создаем новый экземпляр `WebClient` для целевого `serviceUri` (1), выполняем запрос и преобразуем ответ в сущность `User` (2). Когда операция завершается, вручную увеличиваем счетчик с тегом `uri`, значение которого представляет `serviceUri` (3).

Точно так же мы можем объединить обработчики потоков (такие как `.doOnNext()`, `.doOnComplete()`, `.doOnSubscribe()` и `.doOnTerminate()`) с различными типами параметров `Micrometer` (таких как `.counter()`, `.timer()` и `.gauge()`) для измерения нужных характеристик реактивных потоков с достаточным уровнем детализации, обеспечиваемым тегами, которые представляют разные измерения.

Распределенная трассировка с Spring Boot Sleuth

Другим важным компонентом успешного функционирования реактивной системы является понимание того, как протекают события через службы, как обрабатываются запросы и сколько времени это занимает. Как отмечалось в главе 8 «Масштабирование с *Cloud Streams*», обмен данными между службами является важной частью работы распределенной системы, и некоторые проблемы нельзя решить без полного представления о потоках данных. Однако, принимая во внимание всю сложность взаимодействий в распределенной системе, добиться полного понимания этого вопроса часто очень и очень сложно. К счастью, `Spring Cloud` предлагает отличный модуль `spring-cloud-sleuth`. `Spring Cloud Sleuth` – это проект, хорошо интегрированный в инфраструктуру `Spring Boot 2.x` и обеспечивающий распределенную трассировку с помощью всего лишь нескольких автоматически настраиваемых параметров.



Узнать больше о `Spring Cloud Sleuth` можно на странице <https://cloud.spring.io/spring-cloud-sleuth/single/spring-cloud-sleuth.html>.

Большинство современных инструментов трассировки, таких как `Zipkin` или `Brave`, не полностью поддерживает новые реактивные веб-службы `Spring`, но `Spring Cloud Sleuth` естественным образом восполняет этот пробел.

Прежде всего существует `org.springframework.cloud.sleuth.instrument.web.TraceWebFilter` – идентичная замена фильтров `WebMvc` для `WebFlux`. Этот фильтр гарантирует тщательный анализ всех входящих HTTP-запросов и передачу в `Zipkin` всех обнаруженных заголовков трассировки. Также `org`.

`springframework.cloud.sleuth.instrument.web.client.TracingHttpClientInstrumentation` добавляет заголовки трассировки во все исходящие запросы. Для трассировки исходящих запросов мы должны зарегистрировать `WebClient` как контекстный компонент, а не создавать его каждый раз, иначе заголовки добавляться не будут.

Также мы должны выяснить, как хранятся и передаются все данные трассировки в парадигме реактивного программирования. Как мы уже знаем из предыдущих глав, реактивное программирование не гарантирует выполнения всех преобразований в одном потоке выполнения. Это означает, что передача метаданных через `ThreadLocal`, основной шаблон трассировки для **WebMVC**, здесь действует не так, как в парадигме императивного программирования. Однако благодаря особенностям контекста `Context` из библиотеки `Reactor` (описанного в главе 4 «*Project Reactor – основа реактивных приложений*»), глобальных обработчиков `Hooks` и `org.springframework.cloud.sleuth.instrument.web.client.TraceExchangeFilterFunction`, возможно организовать передачу дополнительных контекстных метаданных через реактивный конвейер без `ThreadLocal`.

Наконец, `Spring Cloud Sleuth` предоставляет несколько способов передачи собранной информации о трассировке на сервер `Zipkin`. Самый распространенный способ – через `HTTP`. К сожалению, на данный момент этот метод реализован блокирующим образом. Однако по умолчанию `zipkin2.reporter.AsyncReporter` выполняет отправку данных трассировки на сервер `Zipkin` в отдельном потоке выполнения. Поэтому, несмотря на блокирующую природу, эта технология может быть достаточно эффективной даже для реактивных приложений, так как вызывающая реактивная сторона защищена от задержек, вызванных блокировками, и потенциальных исключений.

Наряду с традиционным протоколом доставки данных по `HTTP` имеется возможность использовать очереди сообщений. `Spring Cloud Sleuth` обеспечивает отличную поддержку `Apache Kafka` и `RabbitMQ`. Хотя клиент для `Apache Kafka` поддерживает асинхронную неблокирующую передачу сообщений, базовый механизм по-прежнему использует тот же `AsyncReporter`, который осуществляет взаимодействия с блокирующим способом.

`Spring Cloud Sleuth` предлагает поддержку распределенной трассировки для `Reactive Spring WebFlux` и `Spring Cloud Streams`. Более того, чтобы начать использовать этот замечательный инструмент в своем проекте, вам нужно лишь добавить следующие зависимости (конфигурация `Gradle`).

```
compile('org.springframework.cloud:spring-cloud-starter-sleuth')
compile('org.springframework.cloud:spring-cloud-starter-zipkin')
```

Кроме того, опираясь на зависимости проекта и доступное окружение, `Spring Boot Autoconfiguration` подготавливает все компоненты, необходимые для распределенной трассировки во время выполнения.



При наличии брокеров сообщений в зависимостях, таких как `org.springframework.amqp:spring-rabbit`, автоматически выбираемые конфигурации связи через брокеры сообщений выглядят предпочтительнее, чем через HTTP. Для случаев, когда отправлять данные трассировки предпочтительнее через HTTP, существует свойство с именем `spring.zipkin.sender.type`, которое принимает одно из следующих значений: `RABBIT`, `KAFKA`, `WEB`.

Пользовательский интерфейс Spring Boot Admin

Admin 2.x

С самого начала главной идеей проекта **Spring Boot Admin** (SBA) было создание удобного интерфейса администратора для мониторинга приложений Spring Boot и управления ими. Spring Boot Admin выделяется красивым и удобным пользовательским интерфейсом, который построен на основе конечных точек Spring Actuator. Он открывает доступ ко всей необходимой оперативной информации, такой как работоспособность приложения, показатели загрузки процессора и памяти, флаги запуска JVM, путь к классам приложений, параметры приложений, трассировка HTTP-запросов и события аудита. Он также позволяет проверять и удалять активные сеансы (с помощью `spring-session`), управлять уровнями журналирования, создавать дампы потоков выполнения и кучи и т. д., как показано на рис. 10.4.

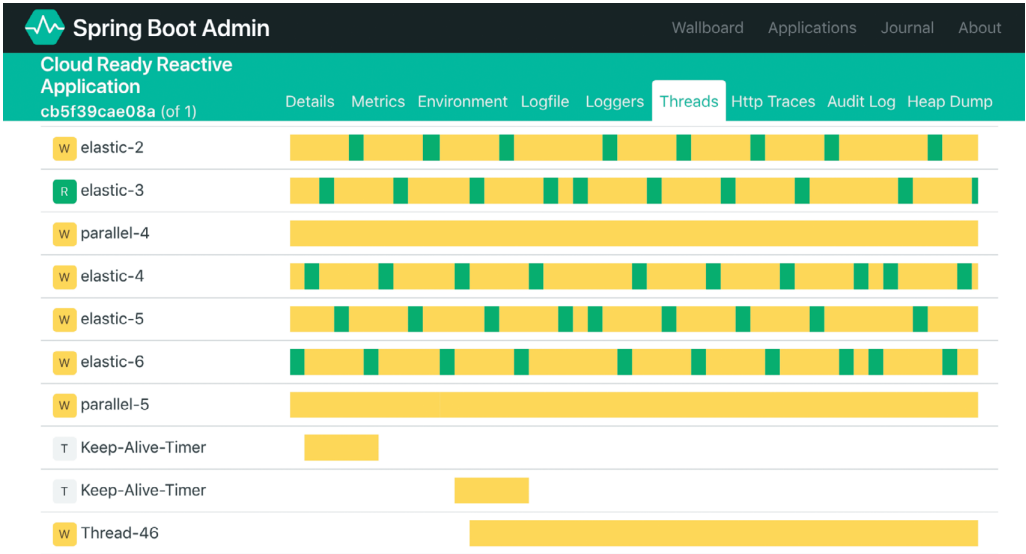


Рис. 10.4. Пользовательский интерфейс Spring Boot Admin, отображающий состояние потоков выполнения в режиме реального времени

Spring Boot Admin создавался для работы с приложениями на основе микросервисов и состоит из двух частей:

- серверная часть действует как центральная точка сбора информации со всех имеющихся микросервисов. Также серверная часть реализует пользовательский интерфейс, через который отображается собранная информация;
- клиентская часть выполняется внутри каждой службы и регистрирует службу на сервере Spring Boot Admin.

Несмотря на то что серверная часть Spring Boot Admin проектировалась как отдельное приложение (рекомендуемая конфигурация), эту роль можно назначить одной из существующих служб. Чтобы назначить службу Spring Boot Admin сервером, достаточно включить в приложение зависимость `de.codecentric:spring-boot-admin-starter-server` и добавить аннотацию `@EnableAdminServer`, чтобы обеспечить необходимую автоматическую настройку. В свою очередь, чтобы интегрировать все соседние службы с сервером Spring Boot Admin, в них следует включить зависимость `de.codecentric:spring-boot-admin-starter-client` и связать с сервером SBA (Spring Boot Admin) или использовать на SBA существующую инфраструктуру Spring Cloud Discovery на основе Eureka или Consul. Также можно создать статическую конфигурацию на стороне сервера SBA.

Сервер SBA поддерживает также репликацию в кластере, поэтому он не должен стать единой точкой отказа в высокодоступном приложении на основе микросервисов (см. рис. 10.5).

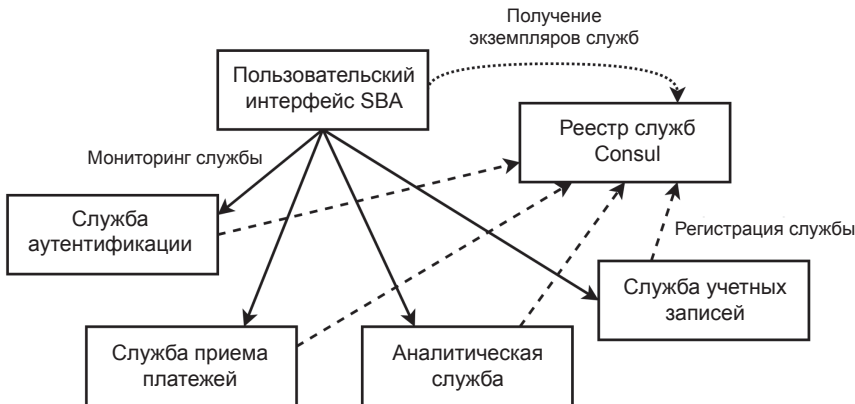


Рис. 10.5. Службы обнаружения и мониторинга Spring Boot Admin

При правильной настройке дашборда Spring Boot Admin он отображает таблицу с зарегистрированными экземплярами служб. Для каждой службы отображается ее состояние работоспособности, что позволяет одним взглядом оценить состояние всей системы. Более того, Spring Boot Admin поддерживает push-уведомления для Slack, Hipchat, Telegram и других служб обмена мгновенными сообщениями.

ми и может информировать оперативный персонал о существенных изменениях в инфраструктуре приложений, а кроме того, даже быть частью инфраструктуры Pager Duty.

До версии 2.0, подобно многим другим расширениям для Spring, Spring Boot Admin основывался на Servlet API. То есть клиентские и серверные приложения использовали блокирующий ввод/вывод и не могли обеспечить высокую эффективность, необходимую реактивным системам. В версии 2.0 сервер SBA был переписан с нуля и теперь использует Spring WebFlux и все взаимодействия осуществляется через асинхронный неблокирующий ввод/вывод.

Разумеется, Spring Boot Admin предоставляет возможности для адекватной защиты пользовательского интерфейса сервера SBA, потому что тот может отображать некоторую конфиденциальную информацию. Spring Boot Admin также обеспечивает интеграцию с Pivotal Cloud Foundry. В случае когда сервер или клиент Spring Boot Admin обнаруживает наличие облачной платформы, применяются все необходимые конфигурации. Соответственно, обнаружение клиентов и серверов упрощается. Также SBA имеет точку расширения, благодаря чему в инфраструктуру SBA можно добавить свое желаемое поведение. Например, можно добавить страницу управления для включения/выключения разных возможностей или настройки аудита.



Узнать больше о Spring Boot Admin можно по ссылке <http://codecentric.github.io/spring-boot-admin/current>.

Подводя итог, отметим, что Spring Boot Admin предоставляет отличную возможность команде DevOps легко отслеживать, эксплуатировать и развивать реактивную систему с помощью удобного и настраиваемого пользовательского интерфейса и пользоваться при этом всеми преимуществами асинхронного и неблокирующего ввода/вывода.

Развертывание в облаке

Самым интересным периодом в жизненном цикле программного обеспечения является его разработка, а самым важным – **эксплуатация**. Имеется в виду обслуживание фактических запросов клиентов, выполнение бизнес-процессов и совершенствование отрасли в целом путем предложения качественных бизнес-услуг. Для нашего приложения этот жизненно важный шаг называется **выпуском**. Выйдя из-под опеки разработчиков и тестировщиков, программная система оказывается лицом к лицу с реальным миром и реальной производственной средой, у которых свои сложности.

Учитывая, насколько сильно могут отличаться целевые окружения и насколько необычными могут быть потребности клиентов или требования к качеству программного продукта, отметим, что сам процесс выпуска часто оказывается слож-

ным и запутанным и потому заслуживает нескольких отдельных книг. Здесь мы не будем рассматривать темы, связанные с поставкой программного обеспечения, а лишь кратко коснемся различных вариантов его развертывания и их влияния на реактивное приложение.

Несмотря на то что благодаря буму IoT (интернета вещей) наши реактивные приложения в скором времени могут оказаться на носимых устройствах, таких как смарт-часы или датчики сердечного ритма, здесь рассмотрим лишь наиболее распространенные целевые окружения, локальные и облачные. Изобретение облачных вычислений изменило подходы к развертыванию приложений и управлению ими. В некотором смысле облачные вычисления заставили нас более *реактивно* относиться к потребностям пользователей, поскольку появилась возможность приобретать новые вычислительные ресурсы в течение нескольких минут, а иногда даже секунд, а не недель или месяцев, как это было в эпоху локальных вычислительных центров. Тем не менее с точки зрения разработчика оба этих подхода довольно похожи. Основное отличие состоит в том, что в последнем случае фактические серверы расположены в здании, а в первом размещаются у поставщика облачных услуг.

Более существенные различия проявляются, когда мы начинаем использовать вычислительные ресурсы для развертывания приложений, и это различие касается развертывания и в локальном, и в облачном окружениях (см. рис. 10.6).

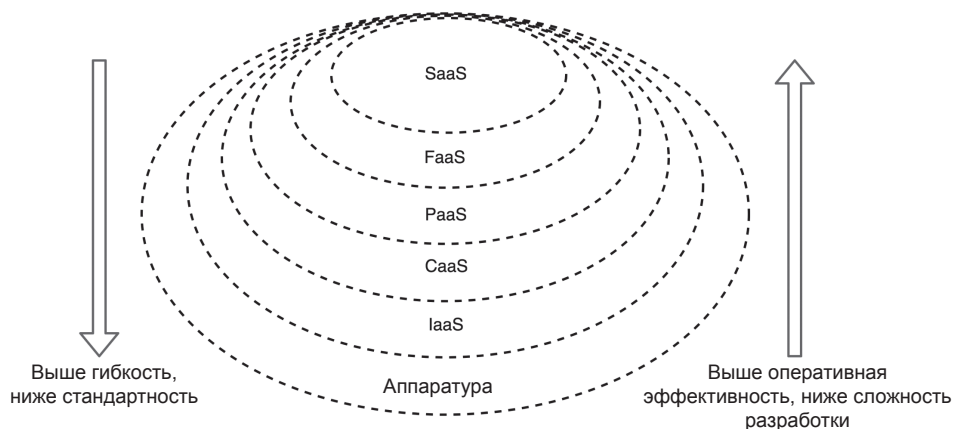


Рис. 10.6. Иерархия платформ и вариантов развертывания программного обеспечения

Вот некоторые пояснения к диаграмме на рис. 10.6.

- **аппаратура:** фактическая серверная аппаратура, которая действует не под управлением гипервизора и, соответственно не виртуализируется, но все еще может предлагаться как облачная услуга. Примерами могут служить Scaleway (<https://www.scaleway.com>) и Packet (<https://www.packet.net>). Пользователь получает в свое распоряжение «голое железо» и должен сам

создавать программное окружение. Даже притом что в этом случае отсутствует гипервизор, на сервер обычно устанавливается какая-то ОС. С точки зрения разработчика этот вариант мало чем отличается от IaaS;

- **IaaS (Infrastructure as a Service – инфраструктура как услуга):** поставщики этого класса услуг предлагают виртуальные машины с комплексными услугами и подключенным хранилищем. В этом случае пользователь должен управлять окружением и развертывать приложения. Варианты развертывания выбираются самим пользователем. Примерами могут служить AWS (<https://aws.amazon.com>) и Microsoft Azure (<https://azure.microsoft.com>);
- **CaaS (Containers as a Service – контейнеры как услуга):** это форма виртуализации, когда облачный провайдер позволяет клиентам развертывать контейнеры (например, контейнеры Docker или CoreOS Rocket). Контейнерная технология определяет режим развертывания, но внутри контейнера клиент может использовать любой технологический стек. Примерами могут служить AWS Elastic Container Service (<https://aws.amazon.com/ecs>) и Pivotal Container Service (<https://pivotal.io/platform/pivotal-container-service>);
- **PaaS (Platform as a Service – платформа как услуга):** поставщики этого класса услуг предлагают все для приложения, вплоть до времени выполнения и построения конвейеров, но при этом ограничивают набор доступных технологий и библиотек. Примерами могут служить Heroku (<https://www.heroku.com>) и Pivotal Cloud Foundry (<https://pivotal.io/platform>);
- **FaaS (Function as a Service – функция как услуга):** недавняя разработка. Поставщики этого класса услуг управляют всей инфраструктурой, а пользователи разворачивают и используют простые функции преобразования данных, которые, как ожидается, будут запущены в течение миллисекунд для обработки отдельных запросов. Примерами могут служить AWS Lambda (<https://aws.amazon.com/lambda>) и Google Cloud Function (<https://cloud.google.com/functions>);
- **SaaS (Software as a Service – программное обеспечение как услуга):** в этом случае все управляется поставщиком услуги. Конечные пользователи могут только пользоваться имеющимися и не могут развертывать свои приложения. Примерами могут служить Dropbox (<https://www.dropbox.com>) и Slack (<https://slack.com>).



Узнать больше о различиях между *IaaS*, *PaaS* и *SaaS* можно в статье <http://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose>. Краткое описание CaaS можно найти по ссылке: <https://blog.docker.com/2016/02/containers-as-a-service-caas>, а вводную статью о FaaS – по ссылке: <https://stackify.com/function-as-a-service-serverless-architecture>.

Реактивные приложения, созданные с использованием стека Spring, можно развертывать в любой среде из перечисленных выше, кроме SaaS, так как она не

поддерживает развертывания пользовательских приложений. Потенциальные варианты запуска реактивных приложений в виде FaaS описываются в главе 8 «*Масштабирование с Cloud Streams*». Там также описаны доступные варианты развертывания приложений Spring в средах PaaS, SaaS и IaaS. Основное различие между этими целевыми средами заключается в формате развертываемых артефактов и в способе создания таких артефактов.

Все еще сохраняется возможность создавать сложные дистрибутивы JVM-приложений для ручной установки. Подобное программное обеспечение можно быстро установить с помощью средств автоматизации, таких как Chef (<https://www.chef.io/chef>) или Terraform (<https://www.terraform.io>), однако данные варианты развертывания не очень привлекательны, когда требуется быстро запустить приложение в облаке, поскольку приводят к дополнительным эксплуатационным расходам.

На настоящий момент наиболее универсальной, пожалуй, единицей развертывания для приложений на Java является uber-jar, также известный как «толстый» (fat) jar. В одном файле этого формата содержится не только программа на Java, но и все ее зависимости. Несмотря на то что uber-jar обычно увеличивает накладные расходы на распространение приложения, особенно если имеется большое количество служб, он все же удобен, прост в использовании и достаточно универсален. Большинство других вариантов распространения построено на основе подхода uber-jar. Существует плагин spring-boot-maven-plugin для Maven, который создает uber-jar с приложением Spring Boot, и spring-boot-gradle-plugin с теми же возможностями для Gradle.



Узнать больше о создании выполняемых jar-файлов с помощью Spring Boot можно в статье <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-executable-jar>. Кроме того, в главе 8 «*Масштабирование с Cloud Streams*» описывается плагин spring-boot-thin-launcher (<https://github.com/dsyer/spring-boot-thin-launcher>) для Maven и Gradle, помогающий уменьшить размеры артефактов, что очень важно в случае развертывания в среде FaaS.

Развертывание в Amazon Web Services

Amazon Web Services (AWS) предлагает несколько вариантов развертывания программного обеспечения, но в основе услуги лежит модель IaaS. Эта модель накладывает минимальное количество ограничений, поэтому мы можем развернуть приложение в AWS десятком способов, но для краткости рассмотрим только вариант создания полностью подготовленных образов, которые можно запускать непосредственно в AWS или VirtualBox.

Проект Boxfuse (<https://boxfuse.com>) делает именно то, что нам нужно: генерирует минимальные образы для приложений Spring Boot, которые загружаются и выполняются непосредственно на виртуальном оборудовании. Boxfuse принимает uber-jar и упаковывает его в минимальный образ виртуальной машины. Он отлично интегрируется с Spring Boot и может использовать информацию из файлов конфигурации Spring Boot для настройки портов и проверки работоспособности. Кроме того, имеется встроенная интеграция с AWS и обеспечивается простая процедура развертывания для приложений Spring Boot.



Описание этапов сборки образа с приложением Spring Boot для AWS можно найти в статье <https://boxfuse.com/blog/spring-boot-ec2.html>.

Развертывание в Google Kubernetes Engine

Google Kubernetes Engine (GKE) – это диспетчер кластеров и управляющая система, построенная компанией Google на базе Kubernetes (<https://kubernetes.io>), системы с открытым исходным кодом для автоматического развертывания, масштабирования и управления контейнерными приложениями. GKE является примером платформы CaaS, и соответственно единицей развертывания в этом случае является образ Docker.

При использовании данного подхода сначала создается uber-jar с приложением, который затем заворачивается в образ Docker. С этой целью можно добавить Dockerfile и использовать команду сборки `docker build` вручную или с помощью плагинов Maven или Gradle, чтобы сделать процедуру частью стандартного конвейера сборки. После тестирования можно развернуть образ в реестре контейнеров Google Cloud Container Registry (<https://cloud.google.com/container-registry>) или использовать службу сборки контейнеров Google Cloud Container Build (<https://cloud.google.com/cloud-build>), чтобы обеспечить фактическое развертывание контейнера.



Описание этапов развертывания приложения Spring Boot в GKE можно найти в статье <https://cloud.google.com/community/tutorials/kotlin-springboot-container-engine>.

Наряду с контейнерами, содержащими наши службы, мы можем развернуть всю инфраструктуру мониторинга, просто ссылаясь на общедоступные образы Docker. Например, можно сослаться на Prometheus, Grafana и Zipkin в одном файле с настройками кластера. Платформы на основе Kubernetes предлагают встроенные механизмы автоматического масштабирования, что позволяет легко добиться эластичности системы.

Аналогично можно развертывать приложения на любой платформе CaaS, в том числе на Amazon Elastic Container Service, Azure Container Service, Pivotal Container

Service, или даже на локальных решениях, таких как OpenShift (<https://www.openshift.com>) и Rancher (<https://rancher.com>).

Развертывание в Pivotal Cloud Foundry

Для развертывания приложений в PaaS Spring Boot предлагает отличную поддержку **Google Cloud Platform** (GCP), **Heroku** и **Pivotal Cloud Foundry** (PCF). Здесь мы рассмотрим интеграцию с PCF, но развертывание с использованием других вариантов осуществляется аналогично.

Прежде всего, приступая к развертыванию в PCF, важно понимать, как экосистема Spring помогает в развертывании приложений на платформе PaaS.

Предположим, у нас имеется потоковое приложение на основе микросервисов, состоящее из трех основных частей:

- **служба пользовательского интерфейса**, которая реализует пользовательский интерфейс и дополнительно играет роль шлюза;
- **служба хранилища** для хранения потоковых данных и их преобразования в визуальные элементы, чтобы служба пользовательского интерфейса могла рассылать их подписчикам;
- **служба коннекторов**, отвечающая за правильные настройки подключений к источникам данных и передачу событий от источников в службу хранилища.

Все эти службы взаимодействуют друг с другом через очередь сообщений, роль которой в нашем примере играет RabbitMQ. Более того, для сохранения полученных данных приложение использует MongoDB в качестве гибкого и быстрого хранилища данных.

В целом нам нужно развернуть три службы, общающиеся друг с другом через RabbitMQ, и одна из них взаимодействует с MongoDB. Для успешной работы приложения требуется запустить все эти службы. Если использовать IaaS и SaaS, нам придется самим нести ответственность за развертывание и поддержку RabbitMQ и MongoDB. В случае с PaaS поставщик облачных услуг берет эти хлопоты на себя и снижает наши эксплуатационные издержки.

Чтобы развернуть службу в PCF, мы должны установить Cloud Foundry CLI, упаковать службу командой `mvn package`, выполнить вход в PCF и запустить следующую команду:

```
cf push <имя-реактивного-приложения> -p target/app-0.0.1.jar
```

Через несколько секунд после запуска процесса развертывания приложение должно быть доступно по адресу `http://<имя-реактивного-приложения>.cfapps.io`. PCF распознает приложение Spring Boot и выбирает для него наиболее оптималь-

ную конфигурацию, но, конечно, разработчики могут определить свои предпочтения в файле `manifest.yml`.



Узнать больше о развертывании Java-приложений в PCF можно в статье <https://docs.cloudfoundry.org/buildpacks/java/java-tips.html>.

Хорошо, что платформа заботится о запуске приложения и его базовой конфигурации. С другой стороны, PaaS слишком категорична в отношении обнаружения служб и топологии Сети, поэтому мы не можем развернуть все службы на одном сервере или настроить свою виртуальную сеть. Поэтому все внешние службы можно разместить с URI `localhost`. Такое ограничение приводит к потере гибкости конфигурации. К счастью, современные поставщики PaaS предоставляют развернутую информацию о платформе и сведения о дополнениях или подключенных службах. Таким образом, все важные данные можно получить и все необходимые настройки выполнить сразу после этого. Однако нам по-прежнему приходится писать клиентский или определенный инфраструктурный код для интеграции с API конкретного поставщика PaaS. Но все не так плохо, потому что на помощь приходит Spring Cloud Connectors (<https://cloud.spring.io/spring-cloud-connectors/>):

«Spring Cloud упрощает подключение к сервисам и получение возможностей окружения в облачных платформах, таких как Cloud Foundry и Heroku, особенно для приложений Spring».

Spring Cloud Connectors сокращает потребность в специализированном шаблонном коде для реализации взаимодействий с облачными платформами. Здесь мы не будем описывать детали конфигурации службы, так как эта информация приводится на официальной странице проекта. Вместо этого попробуем использовать готовую поддержку реактивности в PCF, а также опишем, что необходимо для запуска реактивного Spring-приложения внутри PaaS.

Обнаружение RabbitMQ в PCF

Вернемся к нашему приложению. Вся наша система основана на асинхронной связи через RabbitMQ. Как рассказывалось в главе 8 «Масштабирование с Cloud Streams», для подключения к локальному экземпляру RabbitMQ нам требуется лишь добавить две дополнительные зависимости Spring Boot. Кроме того, для развертывания в облачной инфраструктуре мы должны добавить зависимости `spring-cloud-spring-service-connector` и `spring-cloud-cloudfoundry-connector`. Наконец, мы должны предоставить конфигурацию, представленную в следующем фрагменте кода, и дополнительную аннотацию `@ScanCloud`.

```
@Configuration
@Profile("cloud")
public class CloudConfig extends AbstractCloudConfig {
```

```

@Bean
public ConnectionFactory rabbitMQConnectionFactory() {
    return connectionFactory().rabbitConnectionFactory();
}
}

```

Для большей гибкости используем аннотацию `@Profile("cloud")`, которая включает конфигурацию RabbitMQ только при работе в облаке, а не локально во время разработки.



Для RabbitMQ в PCF нет необходимости предоставлять дополнительные конфигурации, поскольку Cloud Foundry хорошо интегрируется с экосистемой Spring и все необходимые зависимости внедряются во время выполнения без лишних хлопот. Однако важно придерживаться этой практики (по крайней мере, добавлять аннотацию `@ScanCloud`), чтобы обеспечить совместимость со всеми облачными провайдерами. В случае сбоя развернутого приложения проверьте подключение к службе RabbitMQ, предоставляемой PCF.

Обнаружение MongoDB в PCF

Кроме простой конфигурации RabbitMQ существует несколько более сложная конфигурация для хранилищ реактивных данных. Одним из концептуальных различий между реактивными и нереактивными хранилищами данных является разная реализация клиента и/или драйвера. Нереактивные клиенты и драйверы хорошо интегрированы в экосистему Spring и проверены многими решениями. Реактивные клиенты, напротив, все еще считаются новинкой.

На момент написания этих строк версия PCF 2.2 не предлагала готовой конфигурации для реактивной версии MongoDB (как и для любых других баз данных с реактивными клиентами). К счастью, используя модуль Spring Cloud Connectors, можно получить доступ к необходимой информации и настроить реактивного клиента MongoDB так, как показано в следующем примере кода:

```

@Configuration
@Profile("cloud")
public class CloudConfig extends AbstractCloudConfig {           // (1)
    ...
    @Configuration                                             // (2)
    @ConditionalOnClass(MongoClient.class)                      //
    @EnableConfigurationProperties(MongoProperties.class)        //
    public class MongoCloudConfig
        extends MongoReactiveAutoConfiguration {
        ...
        @Bean

```

```

@Override
public MongoClient reactiveStreamsMongoClient(           // (3)
    MongoProperties properties,
    Environment environment,
    ObjectProvider<List<MongoClientSettingsBuilderCustomizer>>
        builderCustomizers
) {
    List<ServiceInfo> infos = cloud()                     // (3.1)
        .getServiceInfos(MongoDbFactory.class);

    if (infos.size() == 1) {
        MongoServiceInfo mongoInfo =
            (MongoServiceInfo) infos.get(0);
        properties.setUri(mongoInfo.getUri());           // (3.2)
    }
    return super.reactiveStreamsMongoClient(             // (3.3)
        properties,
        environment,
        builderCustomizers
    );
}
}
}

```

Пояснения к коду.

1. Расширяет класс `AbstractCloudConfig`, дающий доступ к MongoDB.
2. Типичный подход к проверке присутствия `MongoClient` в пути поиска классов (classpath).
3. Конфигурация реактивного компонента `MongoClient`, где в строке (3.1) получаем информацию о MongoDB из облачного коннектора, включая URI для подключения (3.2). В строке (3.3) создается новый реактивный поток клиента MongoDB повторным использованием оригинальной логики.

Здесь мы расширили класс конфигурации из `org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration` и динамически настроили `MongoProperties` с учетом доступных конфигураций `cloud()`.

После выполнения инструкций Cloud Foundry на странице <https://docs.run.pivotal.io/devguide/deploy-apps/deploy-app.html> и соответствующей настройки MongoDB служба хранилища должна стать доступной и потоковые данные должны сохраняться в MongoDB.

Развертывание в PCF без конфигурации с помощью Spring Cloud Data Flow

Несмотря на то что Pivotal Cloud Foundry упрощает общий процесс развертывания, а экосистема Spring Cloud оказывает всестороннюю помощь для минимизации объема конфигурации внутри приложения, нам все еще приходится иметь дело с некоторыми настройками. Однако, как рассказывалось в главе 8 «*Масштабирование с Cloud Streams*», существует фантастическое решение, которое упрощает разработку приложений для облачного окружения. Это решение называется Spring Cloud Data Flow. Основная идея проекта – упрощение разработки реактивной системы через удобный интерфейс. Наряду с этой центральной функцией Spring Cloud Data Flow предлагает обширный список реализаций для разных поставщиков облачных услуг, который можно найти по ссылке <https://cloud.spring.io/spring-cloud-dataflow/#platform-implementations>. Наиболее значимыми для наших вариантов использования являются реализации для PCF. Spring Cloud Data Flow можно установить в PCF и получить готовое решение без сервера для развертывания конвейеров непосредственно в PCF.



Всю необходимую документацию по установке Spring Cloud Data Flow в PCF можно найти здесь: <http://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/current/reference/htmlsingle>.

Подводя итог, отметим, что Pivotal Cloud Foundry как PaaS предлагает обширную поддержку упрощения процесса развертывания приложений с минимальными усилиями со стороны разработчиков. Она решает множество инфраструктурных задач для поддержки наших приложений, таких как обеспечение брокеров сообщений или экземпляров базы данных. Кроме того, PCF хорошо интегрируется с Spring Cloud Data Flow. Как следствие разработка реактивных облачных приложений перестает быть бесплотной мечтой разработчиков.

Knative для FaaS на основе Kubernetes и Istio

В середине 2018 года компании Google и Pivotal объявили о запуске проекта Knative (<https://pivotal.io/knative>). Его цель – реализовать в Kubernetes возможность развертывания и запуска бессерверной обработки данных. Для маршрутизации взаимодействий между службами в Knative используется проект Istio (<https://istio.io>), поддерживающий динамическую конфигурацию маршрута, канареечное развертывание, постепенное обновление версии и A/B-тестирование. Одна из целей Knative – реализовать возможность создания частной платформы FaaS в любой облачной системе, где может запускаться Kubernetes (или локально).

Другой проект в составе Pivotal – Project Riff (<https://projectriff.io>) – реализован поверх Knative. Основная идея Project Riff – упаковывание функций в кон-

тейнеры, развертывание в Kubernetes, соединение этих функций с брокерами событий и масштабирование контейнеров с функциями в зависимости от скорости поступления событий. Кроме того, Project Riff поддерживает функции, которые могут обрабатывать потоки в реактивном режиме с поддержкой Project Reactor и Spring Cloud Function.



Knative и Project Riff пока находятся на начальном этапе разработки, тем не менее в следующих статьях вы найдете описание мотивов, побудивших начать разработку этих проектов, идей, лежащих их в основе, и примеры использования: <https://projectriff.io/blog/first-post-announcing-riff-0-0-2-release>, <https://content.pivotal.io/blog/knative-powerful-building-blocks-for-a-portable-function-platform>, <https://medium.com/google-cloud/knative-2-2-e542d71d531d>.

Пошаговые инструкции по использованию Riff найдете в следующей статье: <https://www.sudoinit5.com/post/riff-intro>.

В некоторых случаях Knative и Project Riff могут расширять возможности модулей Spring Cloud Data Flow и Spring Cloud Function, но чаще они являются конкурентами. В любом случае, Knative дает нам еще одну платформу для развертывания реактивных приложений, реализованных в парадигме FaaS.

Советы по успешному развертыванию приложений

Говоря простыми словами, успешный процесс поставки программного обеспечения состоит из множества шагов и действий на этапах планирования и разработки. Очевидно, что без надлежащей инфраструктуры автоматизированного тестирования невозможно выпустить мало-мальски нетривиальные приложения, и такие тесты должны разрабатываться и развиваться вместе с рабочим кодом, от первых прототипов до конца срока службы приложения. Учитывая сложность реальных приложений, отметим, что современные тестовые сценарии должны также содержать наборы для тестирования производительности, чтобы дать возможность оценить пределы и адекватно рассчитывать расходы на инфраструктуру, такую как серверы, сети, хранилища и т.д.

Чтобы упростить доставку продукта, наладить быструю обратную связь и в то же время иметь возможность исправлять проблемы, до того как их заметит конечный пользователь, важно использовать такие методы, как **непрерывная доставка** (Continuous Delivery) и **непрерывное развертывание** (Continuous Deployment). Также важно отслеживать все критически значимые показатели системы, собирать журналы, выявлять ошибки и реагировать на поведение системы, независимо от того, является ли оно желательным. Широкий спектр параметров приложений и отчетов в режиме реального времени позволяет не только получать инфор-

мацию о текущем состоянии, но и создавать инфраструктуры, поддерживающие автоматическое масштабирование, восстановление после сбоев и оптимизацию производительности.

Кроме того, как было показано в этой книге, реактивный стек Spring Framework предоставляет основу для создания очень эффективных приложений, производительность которых, в отличие от императивных аналогов, не ограничивается блокирующим вводом/выводом. Автоматические конфигурации Spring Boot предоставляют проверенные комбинации возможностей, поэтому разработчикам редко приходится вручную настраивать компоненты. Spring Boot Actuator предлагает все необходимое для успешной работы приложения, опираясь на передовой опыт и стандарты. Модули Spring Cloud легко интегрируют приложения Spring с брокерами сообщений (RabbitMQ, Apache Kafka), распределенной трассировкой (Spring Cloud Sleuth и Zipkin) и многими другими механизмами. Более того, проекты сообщества, такие как Spring Boot Admin, удовлетворяют потребности в готовых решениях, касающихся работы программного обеспечения. Экосистема Spring обеспечивает также превосходную поддержку всех популярных вариантов развертывания программного обеспечения, включая IaaS, CaaS, PaaS и FaaS. Эта комбинация из нескольких моделей и методов, описанных ранее и подобранных в правильной пропорции, позволяет проектировать, создавать, эксплуатировать и развивать успешные реактивные системы. Мы надеемся, что данная книга упростит решение перечисленных задач.

Заключение

В этой главе мы рассмотрели вопросы, связанные с доставкой и эксплуатацией программного обеспечения. Мы также перечислили несколько методик и модулей Spring, помогающих упростить выпуск программного продукта и уменьшить трудности, связанные с его эксплуатацией. Spring Boot Actuator помогает организовать идентификацию служб, определение их работоспособности, получение параметров функционирования и информации о конфигурации, отслеживание запросов, динамическое изменение уровней журналирования и многое другое. Spring Cloud Sleuth и Zipkin также обеспечивают распределенную трассировку для систем на основе микросервисов и для реактивных компонентов. Spring Boot Admin 2.x предоставляет уникальный пользовательский интерфейс, отображающий все оперативные параметры в форме диаграмм и выразительных отчетов. Все это значительно упрощает жизнь подразделений, осуществляющих эксплуатацию и дальнейшее развитие систем (DevOps), и позволяет сосредоточиться на бизнес-задачах, поскольку модули и плагины Spring Boot берут всю рутину на себя.

Также мы видели, насколько просто настроить реактивное приложение Spring для работы в облаке, включая режимы работы IaaS, CaaS и PaaS. Мы кратко описали процесс развертывания приложения в системах AWS с Boxfuse, в GKE с Docker и в PCF, которая изначально разрабатывалась для выполнения приложений Spring.

Таким образом, реактивная система на основе стека Spring уже имеет все необходимое не только для эффективного использования ресурсов, но и для успешной работы в облаке.

Указатель

@

@EventListener, 54
Создание приложения, 54

A

all, оператор, 162
Amazon Web Services (AWS), 486
any, оператор, 162
Apache Kafka, 397
Apache OpenWhisk, 410
Apache Pulsar, 397
Apache RocketMQ, 397
Apache YARN, 407
ApplicationEventPublisher, класс, 52
AWS DynamoDB, 308
AWS Lambda, 406, 410
AWS Redshift, 308
AWS S3, 308
Azure Cosmos DB, 309
Azure Functions, 410

B

BCrypt, алгоритм, 246
blockFirst, оператор, 171
blockLast, оператор, 171
Boxfuse 486
buffer, оператор, 165
build.gradle, файл, 144
ByteBuffer 222

C

CaaS (Containers as a Service –
контейнеры как услуга), 484
cache, оператор, 186

cast, оператор, 157
Cloud Foundry 407
Cloud Streams,
масштабирование, 383
collectList, оператор, 160
collectSortedList, оператор, 160
combineLatest, оператор, 164
completable, метод, 88
Completable, тип из RxJava, 149
compose, оператор, 190
concatMapDelayError, оператор, 170
concatMap, оператор, 168
concat, оператор, 164
count, оператор, 71, 161
CrudRepository 373

D

delayElements, оператор, 188
delaySequence, оператор, 188
DevOps, идеология,
важность поддержки, 456
distinctUntilChanged, оператор, 161
distinct, оператор, 161
doOnComplete, метод, 171
doOnEach, метод, 171
doOnNext, метод, 171
doOnSubscribe, метод, 171
doOnTerminate, метод, 171

E

Eclipse Vert.x, фреймворк, 83
elapsed, оператор, 188
elementAt, оператор, 159
Eureka 390

F

FaaS (Function as a Service – функция как услуга), 485
filter, оператор, 70, 158
flatMapDelayError, оператор, 170
flatMapSequentialDelayError, оператор, 170
flatMapSequential, оператор, 168
flatMap, оператор, 168
Flowable, тип из RxJava, 148
Flux, реактивный тип, 145
fromIterable, метод, 177

G

generate, фабричный метод, 174
Gitter API 296
Google BigTable 309
Google Cloud Platform (GCP) 487
Google Cloud SQL 309
Google Kubernetes Engine (GKE) 486
gRPC
и RSocket 429
Guava, библиотека, 53

H

HAProxy/Nginx 384
hasElements, оператор, 162
Heroku 487
Heroku PostgreSQL as a Service 308
Hystrix, библиотека, 81

I

IaaS (Infrastructure as a Service – инфраструктура как услуга), 484
ignoreElements, оператор, 159
index, оператор, 157
interval, оператор, 188

io.netty.buffer.ByteBuf, библиотека, 223
io.projectreactor:reactor-test, библиотека, 192
Iterator, интерфейс, 64

J

JDK 9,
поддержка Reactive Streams, 121
just, метод, 185

K

Knative 492
Kubernetes 407

L

limitRequest, оператор, 184
ListenableFuture, интерфейс, 40

M

map, оператор, 69, 157
Maybe, тип из RxJava, 149
MBassador, библиотека, 53
merge, оператор, 164
Mesos, 407
Micrometer, библиотека, 466, 472
Параметры по умолчанию
в Spring Boot 473
Реализация своих параметров
Micrometer 478
MongoDB, драйвер с поддержкой
Reactive Streams, 131
Mono, реактивный тип, 145

N

NATS, 397
Netflix Ribbon, 388
Netty, 223
Netty, фреймворк, 241

NSQ, 397

О

Observable, тип из RxJava, 148

Observer, интерфейс, 47

onBackPressureBuffer, оператор, 183

onBackPressureDrop, оператор, 183

onBackPressureError, оператор, 183

onBackPressureLast, оператор, 183

onErrorMap, оператор, 181

onErrorResume, оператор, 181

onErrorReturn, оператор, 181

onError, оператор, 181

Р

PaaS (Platform as a Service –
платформа как услуга), 484

Pivotal Cloud Foundry (PCF), 487

обнаружение MongoDB в, 489

обнаружение RabbitMQ в, 489

polyglot persistence (использование
хранилищ разного типа), 306

PooledDataBuffer, 223

Pretty Damn Quick, 269

Processor, интерфейс, 99, 109

Project Reactor, библиотека, 137

версия 1.x, 138

версия 2.x, 141

версия 3.x, 142

добавление в проект, 144

дополнения к, 192

интеграция с Akka, 192

как основа реактивных

приложений, 137

краткая история, 137

независимость от механизма

параллельного выполнения, 144

основы, 142

особенности внутренней
реализации, 210

продвинутые средства, 193

расширения для Netty, 140

слияние операторов, 210

тестирование и отладка, 192

Protobuf, 429

Protocol Buffers, 429

Publisher, интерфейс, 99

publishOn, оператор, 199

PULL-PUSH, модель обмена
данными, 137

PULL-PUSH, модель распространения
данных, 144

PULL, модель обмена данными, 89

PULL, модель распространения
данных, 144

PUSH-PULL, гибридная модель обмена
данными, 105

push, метод, 173

PUSH, модель распространения
данных, 89

R

R2DBC, 367

RabbitMQ, 397

Ratpack, фреймворк,
усовершенствования, 130

ReactiveCocoa, библиотека, 80

ReactiveMongoRepository,
детали реализации, 345

ReactiveMongoTemplate,
использование, 346

Reactive Streams, стандарт, 83, 85

Processor, интерфейс, 99, 109

Publisher, интерфейс, 99

PUSH-PULL, гибридная модель

обмена данными, 105
Subscriber, интерфейс, 99
TCK (Reactive Streams Technology Compatibility Kit), 114
асинхронный и параллельный API, 123
в действии, 106
основные положения, 99
поддержка JDK 9, 121
проверка совместимости, 113
Reactor EventBus, 142
reactor-extra, модуль, 192
reactor-logback, модуль, 192
Reactor-Netty,
и RSocket, 421
Reactor Pattern, шаблон поведения, 138
reduce, оператор, 162
retry, оператор, 181
Ribbon, библиотека, 81
RSocket, библиотека, 383
в Java, 425
в Proteus, 433
в ScaleCube, 432
в Spring Framework, 430
в других фреймворках, 432
и gRPC, 429
и Reactor-Netty, 421
реактивная передача сообщений, 420
rxjava2-jdbc, библиотека, 372
RxJava, реактивный фреймворк, 62
RxNetty, 421
RxNetty, библиотека, 81
RxObserver, интерфейс, 64

S

SaaS (Software as a Service – программное обеспечение как услуга), 485
SAGA, шаблон, 312
sampleTimeout, оператор, 170
sample, оператор, 170
scan, оператор, 162
ServletRequest, интерфейс, 233
share, оператор, 187
single, оператор, 159
Single, тип из RxJava, 148
skipUntilOther, оператор, 159
skip, оператор, 159
sort, оператор, 162
Spring,
Spring Boot 2, библиотека, 216
Spring Boot Actuator, 460
Безопасность конечных точек, 471
и реактивность, 220
конечные точки, 461
первые реактивные решения в, 44
реализация своей конечной точки, 469
Spring Boot Admin, 480
Spring Boot Bill of Materials (BOM), спецификация, 460
Spring Boot Sleuth, 478
Spring Boot Thin Launcher, 414
Spring Boot, библиотека, 216
быстрый старт, 217
Spring Boot, инструмент, 54
Spring Cloud, 34, 228
и реактивность, 227
Spring Cloud Data Flow, 407, 491
дашборд, 418

-
- Spring Cloud Deployer Local, 420
 - Spring Cloud Discovery, 427
 - Spring Cloud Function, 229, 409
 - модульная организация приложений, 409
 - реактивное программирование в облаке, 406
 - Spring Cloud Function Deployer, 414
 - Spring Cloud Netflix Zuul, модуль, 228
 - Spring Cloud Sleuth, модуль, 230
 - Spring Cloud Starter Stream App Function, 416
 - Spring Cloud Streams, 383
 - как мост в экосистему Spring, 397
 - реактивное программирование в облаке, 406
 - Spring Core,
 - и реактивность, 221
 - поддержка преобразования реактивных типов, 221
 - реактивные кодеки, 223
 - реактивный ввод/вывод, 222
 - Spring Data,
 - и реактивность, 226
 - реактивные коннекторы, 361
 - Spring Data Cassandra Reactive, модуль, 226
 - Spring Data Couchbase Reactive, модуль, 227
 - Spring Data Mongo Reactive, модуль, 226
 - Spring Data Redis Reactive, модуль, 227
 - Spring Expression Language (SpEL), 53
 - Spring Framework, 7, 22
 - Spring Integration, 397
 - Spring Message, 397
 - Spring Roo, 219
 - ускорение разработки приложений, 219
 - Spring Security,
 - и реактивность, 228
 - Spring Session,
 - и реактивность, 227
 - Spring Test,
 - и реактивность, 229
 - Spring Tool Suite, 218
 - Spring WebFlux, модуль, 224
 - SSE (Server-Sent Events – серверных событий поток), 54
 - StepVerifier, 436
 - знакомство, 436
 - продвинутые приемы тестирования, 440
 - Subject, интерфейс, 47
 - subscribeOn, оператор, 202
 - Subscriber, интерфейс, 99
 - subscribe, оператор, 181
 - Subscription, интерфейс, 99
 - switchIfEmpty, оператор, 181
- Т**
- takeLast, оператор, 159
 - takeUntilOther, оператор, 159
 - takeUntil, оператор, 159
 - take, оператор, 159
 - TCK, 114
 - проверка Publisher, 115
 - проверка Subscriber, 117
 - Technology Compatibility Kit, 114
 - проверка Publisher, 115
 - проверка Subscriber, 117
 - Test Pyramid, 436
 - thenEmpty, оператор, 163
 - thenMany, оператор, 163

then, оператор, 163

Thymeleaf,

поддержка WebFlux, 256

timeout, оператор, 188

timestamp, оператор, 158, 188

tolerable, метод, 171

toStream, метод, 171

transform, оператор, 189

U

using, фабричный метод, 177

V

Vert.x, библиотека,

усовершенствования, 129

W

WebClient, 225

неблокирующие

взаимодействия, 246

WebFlux, модуль, 231, 285

взаимодействия с другими

реактивными библиотеками, 262

и Akka Streams, 262

и RxJava, 262

и медленные соединения, 294

и микросервисы, 292

и потоковые системы, 294

и системы реального времени, 294

как основа реактивного сервера, 231

модели обработки, 272

неблокирующие взаимодействия
с WebClient, 246

основные элементы реактивной
веб-инфраструктуры, 237

поддержка Thymeleaf, 256

потребление памяти, 285

практический пример, 295

проблемы модели обработки, 282

пропускная способность

и задержка, 274

реактивные механизмы

шаблонов, 255

сравнение WebFlux WebSocket
и Spring WebSocket, 252

сравнение с Web MVC, 263, 272

сравнение с WebMVC в форме
конвейеров, 225

сценарии практического
применения, 292

удобство использования, 291

функциональные приемы, 242

функциональный API, 244

WebFlux, тестирование, 445

WebMVC, модуль, 225

сравнение с WebFlux в форме
конвейеров, 225

WebSocket, 251

сравнение WebFlux WebSocket
и Spring WebSocket, 252

тестирование, 451

WebTestClient, тестирование
контроллеров, 446

windowUntil, оператор, 165

window, оператор, 165

Z

Zipkin, 459

zip, оператор, 71, 164

Zuul, библиотека, 81

А ад обратных вызовов, 142

Амдала, закон, 265

асинхронное выполнение,
настройка поддержки, 59

асинхронные взаимодействия,
с Spring Web MVC, 57

асинхронные драйверы (Cassandra), 350
асинхронный доступ к базам данных, 365

Б

база данных как услуга, 307
AWS DynamoDB, 308
AWS Redshift, 308
AWS S3, 308
Azure Cosmos DB, 309
Google BigTable, 309
Google Cloud SQL, 309
Heroku PostgreSQL as a Service, 308

базы данных,
асинхронные и неблокирующие взаимодействия, 90
асинхронный доступ, 365
доступ к, 301
драйвер, 316, 318
реляционное реактивное соединение с, 367

балансировка нагрузки,
на стороне клиента, 386
на стороне сервера, 384
с Spring Cloud и Ribbon, 386
с применением очереди сообщений, 392

«Банда четырех» (Gang of Four, GoF), 45

Бен Кристенсен (Ben Christensen), 80

бесконфликтно реплицируемые типы данных, 314

брокер событий, 53

брокер сообщений, 28

брокеры сообщений, 384, 392
и Spring Cloud Streams, 402
рынок, 396
эластичность, 395

буферизация элементов, 164

В

взаимодействия на основе обмена сообщениями, 25
виртуальное время, 442
время, 188
встроенные базы данных, 316
выпуск, 456

Г

генерация последовательности асинхронных событий, 68
глубоко вложенный код, 142
группировка 164

Д

драйвер базы данных, 316

Ж

жизненный цикл реактивных потоков данных, 194
этап выполнения, 196
этап подписки, 195
этап сборки, 194

З

законы сравнения фреймворков, 264
закон Амдала, 265
закон Литтла, 264
универсальный закон масштабируемости, 269

И

извлечение выборки элементов, 170
изоляция, 25
использование хранилищ разного типа, 306

К

кадрирование, 164
 канал передачи событий, 53
 кеширование элементов потока, 186
 комбинирование реактивных потоков, 164
 комбинирование реактивных технологий, 133
 компонент, 25
 компоновка и преобразование реактивных потоков, 189
 конвейер обработки платежей, 407
 конечные точки,
 безопасность, 471
 для получения информации о параметрах работы, 466
 для получения информации о работоспособности, 463
 для получения информации о службе, 461
 для управления журналированием, 467
 другие, важные, 468
 реализация для Actuator, 469
 контейнеры сервлетов, 233
 кортеж, структура данных, 158
 краткая история реактивных библиотек, 80

Л

Лесли Лэмпорт (Lasley Lamport),
 ссылка на статью, 51
 Литтла, закон, 264

М

Майк Янгстрем (Mike Youngstrom), 219
 манифест реактивных систем, 29

Мартин Фаулер (Martin Fowler),
 ссылка на статью, 47
 масштабирование, 383, 486, 492
 масштабируемости,
 универсальный закон, 269
 механизм обратного давления, 101
 микросервисы, разделение данных между, 309
 микросервисы, хранение данных, 303
 модели обработки данных, 302
 модели симметричных взаимодействий, 424
 мониторинг
 добавление в проект, 460
 и реактивность, 229
 планировщиков в Reactor, 475
 получение информации о параметрах работы, 466
 получение информации о работоспособности, 463
 получение информации о службе, 461
 потоков в Reactor, 474
 реактивных Spring-приложений, 460
 реактивных потоков данных, 474
 реализация конечной точки для Actuator, 469
 управление журналированием, 467

Н

неблокирующие операции, 27
 неблокирующий обмен сообщениями, 28
 Нил Гюнтер (Neil Gunther), 269

О

обертывание транзакций, 178
 обработка ошибок, 180

обратное давление, 33, 183
объектно-реляционное
 отображение, 325
ограниченные контексты, 302
одноразовые ресурсы, передача
 в реактивные потоки, 175
оптимистическая репликация, 314
основные элементы реактивной
 веб-инфраструктуры, 237
отказ, 25
отображение элементов реактивных
 последовательностей, 157
очереди сообщений, 392
ошибки, обработка, 180

П

пакетная обработка элементов
 потока, 164
первые реактивные решения
 в Spring, 44
планировщики, 205
подписка на реактивный поток, 151
подписчики, реализация, 154
поиск подходящего оператора, 173
пользовательский интерфейс
 с поддержкой SSE, 60
последовательности Flux и Mono, 149
потоки данных, холодные
 и горячие, 184
потоки, производство и потребление, 65
предметно-ориентированное
 проектирование, 302
преимущества соглашений перед
 конфигурацией, 219
преобразование реактивного
 ландшафта, 126
 Ratpack, 130
 RxJava, 126

Vert.x, 129
преобразование потоков, 69
 оператор count, 71
 оператор filter, 70
 оператор map, 69
 оператор zip, 71
преобразование реактивных
 последовательностей в блокирующие
 структуры, 170
преобразование реактивных
 последовательностей с помощью
 операторов, 156
принципы реактивного дизайна, 54
проблемы несовместимости API, 86
 реализации Observable, 88
пропускная способность, 24
протокол связи для доступа к базе
 данных, 316
процесс автоматической передачи
 данных источником, 94
процессоры, 191
 асинхронные, 191
 непосредственные, 191
 синхронные, 191
процесс получения данных пакетами
 по запросу, 93
процесс получения данных
 по запросу, 91
пул соединений, 320

Р

развертывание,
 в Amazon Web Services, 486
 в Google Kubernetes Engine, 486
 в облаке, 483
 развертывание в PCF без
 конфигурации, 491
 советы по успешному развертыванию

- hr/>
- приложений, 492
 - развертывание в Pivotal Cloud Foundry, 487
 - разделение данных между микросервисами, 309
 - разделение ответственности на команды и запросы, 313
 - распределенная трассировка, 478
 - распределенные транзакции, 310
 - бесконфликтно реплицируемые типы данных, 314
 - регистрация событий, 312
 - событийно-ориентированные архитектуры, 310
 - согласованность в конечном счете, 311
 - шаблон SAGA, 312
 - распространение данных в реактивном потоке, 143
 - реактивная безопасность, 258
 - использование, 261
 - реактивная передача сообщений с низкой задержкой, 420
 - реактивное веб-ядро, 234
 - реактивное программирование, 22
 - основные понятия, 44
 - реактивное программирование в облаке, 406
 - реактивное соединение с реляционной базой данных, 367
 - реактивность, 22
 - в Spring, 33
 - в Spring Boot 2.0, 220
 - в Spring Cloud, 228
 - в Spring Core, 221
 - в Spring Data, 226
 - в Spring Security, 228
 - в Spring Session, 227
 - в Spring Test, 229
 - в Web, 224
 - в мониторинге, 229
 - основные преимущества, 22
 - примеры использования, 30
 - реактивные библиотеки,
 - краткая история развития, 80
 - реактивные драйверы (MongoDB), 348
 - реактивные кодеки, 223
 - реактивные коннекторы, 361
 - Cassandra, 362
 - Couchbase, 362
 - MongoDB, 361
 - Redis, 363
 - реактивные механизмы шаблонов, 255
 - реактивные потоки данных,
 - мониторинг, 474
 - реактивные потоки, тестирование, 435
 - тестирование с StepVerifier, 436
 - реактивные приложения,
 - виртуальное время, 442
 - мониторинг, 460
 - получение информации о параметрах работы, 466
 - получение информации о службе, 461
 - получение сведений о работоспособности, 463
 - развертывание в облаке, 483
 - тестирование, 435
 - управление журналированием, 467
 - реактивные системы, 22
 - реактивные типы из RxJava, 148
 - реактивные транзакции, 352
 - в MongoDB, 352
 - реактивные фреймворки, 238
 - реактивные хранилища, 344
-

поддержка разбиения на
страницы, 345

реактивный Spring Data, 378

реактивный WebSocket API, 249

 клиентский WebSocket API, 251

 серверный WebSocket API, 250

реактивный доступ в Spring Data, 334

 MongoDB, 336

 объединение операций
 с хранилищами, 339

реактивный доступ
 к SecurityContext, 258

реактивный доступ
 к базам данных, 301

реактивный контекст, тестирование, 445

реактивный ландшафт, 81

реактивный поток SSE и легковесная
 замена WebSockets, 253

реализация своих подписчиков, 154

регистрация событий, 312

ресурс, 27

С

сбор данных из реактивных
 последовательностей, 160

серверных событий поток
 (Server-Sent Events, SSE), 253

сигналы, материализация
 и дематериализация, 172

синхронная модель извлечения
 данных, 316

 JDBC, 319

 Spring Data JDBC, 323

 Spring JDBC, 322

 реактивный доступ, 321

 управление соединениями, 320

 JPA, 326

 Spring Data JPA, 327

 реактивность, 328

 NoSQL, 329

 достоинства, 333

 драйвер базы данных, 318

 ограничения, 332

системы, управляемые
 сообщениями, 384

событийно-ориентированные
 архитектуры, 310

согласованность в конечном счете, 311

Т

тестирование, 435

 виртуальное время 442

 реактивного контекста, 445

 реактивных приложений, 435

транзакции, распределенные, 310

транзакции, реактивные, 352

У

управление обратным давлением, 143

управления потоком данных, 95

 быстрый производитель, медленный
 потребитель, 95

 медленный производитель, быстрый
 потребитель, 95

 неограниченная очередь, 96

 ограниченная очередь
 с блокировкой, 97

 ограниченная очередь со сбросом
 элементов, 96

Ф

фильтрация реактивных
 последовательностей, 158

функции как служба (Function as a
 Service, FaaS), 229

функциональный API обработки
запросов и ответов, 244

Х

хранение данных в микросервисах, 303

Ц

цепочки из операторов, 143

Ш

шаблоны 52

«Итератор», 63

«Наблюдатель», 45, 49

«Публикация/Подписка», 44, 52

шина событий, 53

широковещательная рассылка элементов
потока данных, 185

Э

эластичность, 24, 25

Я

язык описания интерфейса, 429

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru.**

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru.**

Олег Докука
Игорь Лозинский

Практика реактивного программирования в Spring 5

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Киселев А. Н.*

Корректоры *Юрьева В. И., Синяева Г. И.*

Верстка *Орлов И. Ю.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «PT Serif».

Печать цифровая. Усл. печ. л. 41,28.

Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**