

Осваиваем Kubernetes

Джиджи Сайфан

Оркестрация контейнерных архитектур



Packt>

Mastering Kubernetes

Second Edition

Master the art of container management by using the power of Kubernetes

Gigi Sayfan

Packt>

BIRMINGHAM - MUMBAI

Осваиваем Kubernetes

ОРКЕСТРАЦИЯ КОНТЕЙНЕРНЫХ АРХИТЕКТУР

Джиджи Сайфан



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

ББК 32.973.23-018
УДК 004.45
С14

Сайфан Джиджи

С14 Осваиваем Kubernetes. Оркестрация контейнерных архитектур. — СПб.: Питер, 2019. — 400 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0973-9

В книге рассмотрена новейшая версия Kubernetes 1.10.

Kubernetes — это система с открытым кодом, предназначенная для работы с контейнерными приложениями, их развертывания, масштабирования, управления ими. Если вы хотите запустить дополнительные контейнеры или автоматизировать управление, то вам не обойтись без Kubernetes.

Книга начинается с изучения основ Kubernetes, архитектуры и компоновки этой системы. Вы научитесь создавать микросервисы с сохранением состояния, ознакомитесь с такими продвинутыми возможностями, как горизонтальное автомасштабирование подов, выкатывание обновлений, квотирование ресурсов, обустроите долговременное хранилище на бэкенде. На реальных примерах вы исследуете возможности сетевой конфигурации, подключение и настройку плагинов. Эта книга поможет вам стать искусным дирижером и обращаться с контейнерными системами любой сложности.

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

ББК 32.973.23-018
УДК 004.45

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1788999786 англ.

ISBN 978-5-4461-0973-9

© Packt Publishing 2018.

First published in the English language under the title «Mastering Kubernetes — Second Edition – (9781788999786)»

© Перевод на русский язык ООО Издательство «Питер», 2019

© Издание на русском языке, оформление ООО Издательство «Питер», 2019

© Серия «Для профессионалов», 2019

Краткое содержание

Об авторе.....	17
О рецензентах.....	18
Предисловие	19
Глава 1. Архитектура Kubernetes	23
Глава 2. Создание кластеров Kubernetes.....	49
Глава 3. Мониторинг, журналирование и решение проблем	71
Глава 4. Высокая доступность и надежность.....	97
Глава 5. Конфигурация безопасности, ограничений и учетных записей в Kubernetes.....	120
Глава 6. Использование критически важных ресурсов Kubernetes.....	149
Глава 7. Работа с хранилищем данных в Kubernetes	177
Глава 8. Запуск приложений с сохранением состояния	209
Глава 9. Плавающие обновления, масштабирование и квоты.....	234
Глава 10. Продвинутая организация сети в Kubernetes.....	264
Глава 11. Запуск Kubernetes в нескольких облаках и многокластерном режиме	301
Глава 12. Настройка Kubernetes: API и дополнения	333
Глава 13. Работа с диспетчером пакетов Kubernetes.....	363
Глава 14. Будущее Kubernetes.....	385

Оглавление

Об авторе.....	17
О рецензентах.....	18
Предисловие	19
Для кого предназначена книга	19
Темы, рассмотренные в книге.....	20
Как извлечь из книги максимальную пользу.....	21
Полноцветные изображения.....	21
Условные обозначения.....	21
Глава 1. Архитектура Kubernetes.....	23
Что такое Kubernetes	23
Чем Kubernetes не является	24
Оркестрация контейнеров	24
Контейнеры на физических и виртуальных устройствах.....	24
Преимущества контейнеров.....	25
Контейнеры в облаке	25
От мелких домашних животных к крупному рогатому скоту	26
Концепции, лежащие в основе Kubernetes.....	26
Кластер	28
Узел.....	28
Ведущий узел	28
Под	28
Метка.....	29
Аннотации	29
Селекторы меток.....	30
Контроллеры репликации и наборы реплик.....	30
Сервисы.....	31
Том	31

StatefulSet	32
Конфиденциальная информация	32
Имена	32
Пространства имен	33
Углубленное рассмотрение архитектуры Kubernetes	33
Шаблоны проектирования распределенных систем	33
API Kubernetes	35
Категории ресурсов	36
Ведущие компоненты Kubernetes	38
API-сервер	38
Etcd	38
Диспетчер контроллеров Kube	38
Диспетчер облачных контроллеров	39
kube-scheduler	40
DNS	40
Узловые компоненты	40
Прокси-сервер	40
Kubelet	41
Среды выполнения, совместимые с Kubernetes	41
Интерфейс среды выполнения контейнеров (CRI)	41
Docker	43
Rkt	44
Контейнеры Nupur	46
Непрерывная интеграция и разработка	46
Цепочка CI/CD	46
Проектирование цепочки CI/CD для Kubernetes	48
Резюме	48
Глава 2. Создание кластеров Kubernetes	49
Быстрое создание одноузлового кластера с помощью Minikube	49
Подготовка	50
В Windows	50
В macOS	50
Создание кластера	51
Отладка	53
Проверка кластера	54
Выполнение работы	54
Исследование кластера с помощью панели управления	55

Создание многоузлового кластера с помощью kubeadm.....	57
Чего следует ожидать	57
Подготовка к работе.....	57
Подготовка кластера виртуальных машин на основе vagrant.....	57
Установка необходимого программного обеспечения.....	58
Создание кластера	61
Настройка pod-сети.....	62
Добавление рабочих узлов.....	63
Создание кластеров в облаке (GCP, AWS и Azure).....	64
Интерфейс cloud-provider	64
Google Cloud Platform	65
Amazon Web Services	65
Azure	66
Alibaba Cloud	67
Создание аппаратного кластера с нуля	68
Сценарии использования «голого железа».....	68
Когда создание аппаратного кластера имеет смысл.....	68
Процесс создания кластера.....	69
Использование инфраструктуры частного виртуального облака.....	69
Резюме.....	70
Глава 3. Мониторинг, журналирование и решение проблем	71
Мониторинг Kubernetes с помощью Heapster	71
Установка Heapster	72
Хранилище InfluxDB.....	74
Структура хранилища	74
Визуализация с помощью Grafana.....	77
Анализ производительности с помощью панели управления.....	78
Представление верхнего уровня.....	78
Добавление централизованного журналирования.....	84
Обнаружение неисправностей на узлах.....	86
Node Problem Detector	86
DaemonSet	87
Демоны для обнаружения проблем.....	87
Примеры потенциальных проблем.....	87
Проектирование устойчивых систем	88
Аппаратные сбои	88

Квоты, общие ресурсы и лимиты	89
Плохая конфигурация.....	90
Соотношение затрат и производительности	91
Использование Prometheus	92
Что такое операторы	92
Prometheus Operator	92
Установка Prometheus с помощью kube-prometheus.....	93
Мониторинг кластера с помощью Prometheus	94
Резюме.....	96
Глава 4. Высокая доступность и надежность.....	97
Концепции, связанные с высокой доступностью.....	97
Избыточность.....	98
Горячая замена.....	98
Выбор лидера.....	98
Умная балансировка нагрузки	99
Идемпотентность	99
Автоматическое восстановление	99
Рекомендуемые методики достижения высокой доступности	100
Создание высокодоступных кластеров	100
Как сделать узлы надежными.....	101
Как обезопасить состояние кластера	101
Сохранность данных	105
Избыточные API-серверы.....	106
Выбор лидера в Kubernetes.....	106
Высокая доступность в тестовой среде	108
Тестирование высокой доступности.....	109
Обновление кластера на лету	110
Плавающие обновления	111
Сине-зеленые обновления	112
Управление изменениями в контрактах данных	113
Миграция данных	113
Устаревание API	114
Производительность, расходы и компромиссы архитектуры крупных кластеров.....	114
Требования к доступности.....	115
Отсутствие гарантий.....	115

Периоды обслуживания	115
Быстрое восстановление	116
Нулевое время простоя.....	116
Производительность и согласованность данных.....	118
Резюме.....	119
Глава 5. Конфигурация безопасности, ограничений и учетных записей в Kubernetes.....	120
Проблемы безопасности, стоящие перед Kubernetes	120
Потенциальные уязвимости узлов	121
Потенциальные уязвимости образов.....	123
Потенциальные проблемы с конфигурацией и развертыванием.....	124
Потенциальные уязвимости подов и контейнеров	125
Потенциальные организационные и культурные проблемы.....	125
Как закаляется Hardening.....	126
Служебные учетные записи в Kubernetes	127
Доступ к API-серверу	128
Защита подов.....	134
Управление сетевыми политиками	139
Использование объектов secret.....	142
Многопользовательские кластеры.....	145
Необходимость в многопользовательских кластерах.....	145
Безопасная мультиарендность на основе пространств имен.....	146
Подводные камни работы с пространствами имен	147
Резюме.....	148
Глава 6. Использование критически важных ресурсов Kubernetes.....	149
Проектирование системы Hue.....	149
Область применения Hue	149
Планирование рабочих процессов.....	153
Использование Kubernetes для построения системы Hue	154
Эффективное применение kubectl.....	154
Файлы конфигурации ресурсов в kubectl.....	155
Развертывание долгоиграющих микросервисов в подах	157
Разделение внутренних и внешних сервисов.....	160
Развертывание внутреннего сервиса.....	161
Создание сервиса hue-reminders.....	162
Выведение сервиса наружу	163

Ограничение доступа с помощью пространства имен	165
Запуск заданий.....	166
Параллельное выполнение заданий	167
Удаление завершенных заданий.....	168
Планирование регулярных заданий с помощью crontab	168
Интеграция с внешними компонентами	170
Компоненты вне сети кластера	170
Компоненты внутри сети кластера.....	170
Управление системой Hue с помощью Kubernetes.....	171
Управление зависимостями с помощью проверок готовности	172
Применение контейнеров инициализации для упорядоченного запуска подов.....	173
Развитие системы Hue с помощью Kubernetes.....	174
Применение Hue на предприятиях.....	175
Двигаем научный прогресс с помощью Hue.....	175
Hue — будущее системы образования	175
Резюме.....	176
Глава 7. Работа с хранилищем данных в Kubernetes	177
Подробное знакомство с постоянными томами	177
Тома	178
Создание постоянных томов.....	182
Запрос постоянного тома	184
Подключение запросов в качестве томов	186
Тома на основе блочных устройств	186
Классы хранилищ	188
Пример работы с постоянным томом от начала до конца	189
Типы томов в облачных хранилищах GCE, AWS и Azure.....	192
AWS Elastic Block Store (EBS)	193
AWS Elastic File System.....	193
Постоянный диск в GCE.....	195
Диски в Azure	195
Файловое хранилище Azure	196
Тома GlusterFS и Ceph в Kubernetes.....	197
Использование GlusterFS	197
Использование Ceph.....	200
Управление томами внутрикластерных контейнеров с помощью Flocker.....	202
Интеграция промышленного хранилища в Kubernetes.....	204

Отображение томов.....	204
Использование сторонних хранилищ с помощью FlexVolume.....	205
Container Storage Interface.....	206
Резюме.....	208
Глава 8. Запуск приложений с сохранением состояния.....	209
Состояние приложений в Kubernetes.....	209
Активная работа с данными в распределенных приложениях.....	209
Зачем управлять состоянием в Kubernetes.....	210
Зачем выносить управление состоянием за пределы Kubernetes.....	210
Механизм обнаружения: общие переменные окружения или DNS-записи.....	211
Обращение к внешним хранилищам данных через DNS.....	211
Обращение к внешним хранилищам данных через переменные окружения.....	211
Использование ConfigMap в виде переменной окружения.....	212
Резервная копия состояния в памяти.....	213
Использование DaemonSet в качестве резервного постоянного хранилища.....	214
Подключение постоянных томов.....	214
Применение StatefulSet.....	214
Выполнение кластера Cassandra в Kubernetes.....	216
Краткое введение в Cassandra.....	216
Docker-образ Cassandra.....	217
Интеграция Kubernetes и Cassandra.....	222
Создание неуправляемого сервиса Cassandra.....	224
Использование контроллера StatefulSet для создания кластера Cassandra.....	225
Распределение Cassandra с помощью контроллера репликации.....	228
Использование DaemonSet для распределения Cassandra.....	232
Резюме.....	232
Глава 9. Плавающие обновления, масштабирование и квоты.....	234
Горизонтальное автомасштабирование подов.....	234
Объявление горизонтального автомасштабирования подов.....	235
Нестандартные показатели.....	237
Автомасштабирование с помощью kubectl.....	238
Плавающие обновления с автомасштабированием.....	240
Ограничение ресурсов с помощью лимитов и квот.....	241
Включение квот на ресурсы.....	242
Типы квот.....	242
Области действия квот.....	244

Запросы и ограничения	245
Работа с квотами	245
Выбор и регулирование мощности кластера	249
Типы узлов	250
Выбор решений для хранения данных	250
Компромисс между денежными затратами и временем отклика	251
Эффективное использование узлов с разной конфигурацией	251
Преимущества эластичных облачных ресурсов	252
Hyper.sh и AWS Fargate в качестве альтернативы	253
Экстремальные нагрузки в Kubernetes	254
Улучшение производительности и масштабируемости Kubernetes	255
Измерение производительности и масштабируемости Kubernetes	258
Тестирование Kubernetes в крупномасштабных кластерах	262
Резюме	263
Глава 10. Продвинутая организация сети в Kubernetes	264
Сетевая модель Kubernetes	264
Взаимодействие между контейнерами внутри пода	264
Взаимодействие между подами	265
Взаимодействие между подами и сервисами	265
Внешний доступ	266
Сетевые возможности Kubernetes и Docker	266
Поиск и обнаружение	268
Сетевые дополнения к Kubernetes	270
Сетевые решения для Kubernetes	276
Создание мостов в аппаратных кластерах	277
Contiv	277
Open vSwitch	278
Nuage Networks VCS	279
Canal	279
Flannel	280
Проект Calico	282
Romana	282
Weave Net	284
Эффективное использование сетевых политик	284
Архитектура сетевой политики в Kubernetes	284
Сетевые политики и CNI-дополнения	285
Конфигурация сетевых политик	285
Реализация сетевых политик	285

Способы балансирования нагрузки.....	286
Внешний балансировщик нагрузки	287
Балансирование нагрузки с помощью внутреннего сервиса	290
Ingress.....	291
Traffic	294
Написание собственного дополнения для CNI	294
Знакомство с дополнением loopback.....	295
Сборка CNI-дополнения на основе готового каркаса.....	297
Обзор дополнения-моста	298
Резюме.....	299
Глава 11. Запуск Kubernetes в нескольких облаках и многокластерном режиме.....	301
Многокластерный режим.....	301
Важные сценарии использования многокластерного режима.....	302
Федеративный управляющий уровень	304
Федеративные ресурсы.....	306
Сложности.....	312
Управление многокластерным режимом Kubernetes.....	316
Настройка многокластерного режима с нуля	316
Начальная настройка.....	317
Использование официального образа Hyperkube.....	317
Запуск федеративного управляющего уровня	317
Регистрация кластеров Kubernetes в федерации.....	318
Обновление KubeDNS	319
Отключение многокластерного режима	319
Настройка многокластерного режима с помощью Kubefed.....	319
Каскадное удаление ресурсов.....	322
Балансировка нагрузки между несколькими кластерами.....	323
Переключение на другие кластеры в случае сбоя	324
Выполнение федеративных рабочих нагрузок	326
Создание федеративного сервиса.....	326
Добавление подов	327
Проверка общедоступных записей DNS.....	327
Обнаружение федеративного сервиса.....	328
Обработка отказов подов и целых кластеров.....	330
Решение проблем	331
Резюме.....	331

Глава 12. Настройка Kubernetes: API и дополнения	333
Работа с API Kubernetes	333
OpenAPI	333
Настройка прокси	334
Непосредственный доступ к API Kubernetes	334
Создание пода с помощью API Kubernetes	337
Доступ к API Kubernetes через клиент Python	338
Расширение API Kubernetes	344
Понимание структуры пользовательского ресурса	344
Определение пользовательских ресурсов	345
Интеграция пользовательских ресурсов	346
Агрегация API-серверов	348
Использование каталога сервисов	349
Написание дополнений Kubernetes	350
Создание пользовательского дополнения-планировщика	350
Проверим, запланированы ли наши поды с помощью пользовательского планировщика	355
Использование веб-хуков для контроля доступа	355
Веб-хуки для аутентификации	355
Веб-хуки для авторизации	357
Веб-хуки для контроля входа	359
Предоставление пользовательских показателей для горизонтального автомасштабирования подов	359
Добавление в Kubernetes пользовательского хранилища	360
Резюме	361
Глава 13. Работа с диспетчером пакетов Kubernetes	363
Знакомство с Helm	363
Преимущества	363
Архитектура	364
Компоненты	364
Использование Helm	365
Установка Helm	365
Поиск схем	367
Установка пакетов	369
Работа с репозиториями	374
Управление схемами с помощью Helm	375

Создание своих собственных схем.....	376
Файл Chart.yaml.....	376
Файлы метаданных схемы.....	377
Управление зависимостями схемы.....	378
Использование шаблонов и значений.....	380
Резюме.....	384
Глава 14. Будущее Kubernetes.....	385
Дорога в будущее.....	385
Версии и этапы развития Kubernetes.....	385
Особые интересы и рабочие группы Kubernetes.....	386
Конкуренция.....	386
Значимость контейнеризации.....	387
Docker Swarm.....	387
Mesos/Mesosphere.....	388
Облачные платформы.....	388
AWS.....	388
Azure.....	389
Alibaba Cloud.....	389
Время Kubernetes.....	390
Сообщество.....	390
GitHub.....	390
Конференции и встречи.....	390
Осведомленность потребителей.....	391
Экосистема.....	391
Провайдеры публичных облаков.....	391
Обучение и подготовка.....	392
Модулирование и дополнения «вне дерева».....	394
Технология service mesh и serverless-фреймворки.....	395
Технология service mesh.....	395
Serverless-фреймворки.....	395
Резюме.....	396

Об авторе

Джиджи Сайфан (Gigi Sayfan) работает главным архитектором программного обеспечения в компании Helix. Вот уже 22 года разрабатывает приложения в сферах обмена мгновенными сообщениями и морфинга¹. Джиджи ежедневно пишет код промышленного уровня на таких языках программирования, как Go, Python, C/C++, C#, Java, Delphi, JavaScript и даже Cobol и PowerBuilder, для операционных систем Windows, Linux, macOS, Lynx (встраиваемые решения) и др. Его знания и опыт распространяются на базы данных, сети, распределенные системы, нестандартные пользовательские интерфейсы и циклы разработки программного обеспечения общего характера.

¹ Технология в компьютерной анимации, визуальный эффект, создающий впечатление плавной трансформации одного объекта в другой. Используется в игровом и телевизионном кино, в телевизионной рекламе. — *Здесь и далее примеч. пер.*

О рецензентах

Дес Друри (Des Drury) — страстный технолог с более чем 25-летним опытом работы в ИТ-индустрии. Дес сумел распознать преимущества платформы Kubernetes на самых ранних этапах ее развития и начал пропагандировать ее всем, кто проявлял малейший интерес. Он соорганизатор Мельбурнской группы пользователей Kubernetes. В 2015 году Дес выпустил собственный дистрибутив Kubernetes под названием Open Data center.

Дес — директор компании Cito Pro, которая специализируется на Kubernetes и программном обеспечении с открытым кодом. В этой должности он помог множеству организаций с внедрением Kubernetes и сопутствующих технологий.

Якуб Павлик (Jakub Pavlik) — сооснователь, бывший технический директор, а ныне главный архитектор компании Tcp Cloud (с 2016 года входит в Mirantis). Вместе со своей командой Якуб несколько лет работал над облачной платформой IaaS, которая обеспечивала развертывание и администрирование проектов OpenStack-Salt и OpenContrail для крупных поставщиков услуг.

В качестве технического директора компании Volterra Inc. он совместно с другими командами опытных профессионалов занимается разработкой и внедрением пограничной вычислительной платформы нового поколения.

Предисловие

Kubernetes — это система с открытым исходным кодом, которая автоматизирует развертывание и масштабирование упакованных в контейнер приложений, а также управление ими. Если вы запускаете много контейнеров или хотите управлять ими автоматически, эта система вам понадобится. Данная книга служит руководством по углубленной работе с кластерами на основе Kubernetes.

Начнем с изучения фундаментальных свойств архитектуры Kubernetes и подробно рассмотрим ее структуру. Вы узнаете, как с помощью системы запускать сложные микросервисы с сохранением состояния, в том числе с применением продвинутых возможностей: горизонтального автомасштабирования подов (pod), выкатывания обновлений, квот на ресурсы и постоянных серверных хранилищ. На реальных примерах исследуете разные варианты сетевой конфигурации и научитесь подготавливать, реализовывать и отлаживать различные сетевые дополнения Kubernetes. В конце книги узнаете, как разрабатывать и использовать нестандартные ресурсы в режиме автоматизации и обслуживания. Это издание затрагивает некоторые дополнительные концепции, появившиеся в версии Kubernetes 1.10, например Prometheus, управление доступом на основе ролей и агрегацию API.

Дочитав до конца, вы овладеете всеми знаниями, необходимыми для перехода со среднего уровня на продвинутый.

Для кого предназначена книга

Эта книга для вас, если вы системный администратор или разработчик со средним уровнем знаний о Kubernetes и хотите овладеть расширенным функционалом этой системы. Кроме того, вы должны иметь базовое представление об устройстве сетей. Материал, который вы здесь найдете, поможет пройти путь к полному освоению Kubernetes.

Темы, рассмотренные в книге

Глава 1 «Архитектура Kubernetes» познакомит вас с устройством системы Kubernetes и поможет понять причины выбора тех или иных архитектурных решений.

Глава 2 «Создание кластеров Kubernetes» представит различные способы создания кластеров Kubernetes. С помощью предложенных инструментов вы разработаете и изучите несколько кластеров.

Глава 3 «Мониторинг, журналирование и решение проблем» поможет подготовить систему мониторинга и измерения в кластерах Kubernetes и понять, как она работает. Это позволит вам обнаруживать и устранять типичные проблемы, с которыми администраторы сталкиваются в ежедневной работе.

Глава 4 «Высокая доступность и надежность» научит проектировать высокодоступные кластеры Kubernetes и выполнять их обновление на лету. Вы научитесь подготавливать свои системы для промышленной среды, для работы в крупных масштабах.

Глава 5 «Конфигурация безопасности, ограничений и учетных записей в Kubernetes» познакомит и научит вас работать с настройками безопасности и ограничений, интеграцией с AAA-серверами, пространствами имен и конфигурацией служебных учетных записей.

Глава 6 «Использование критически важных ресурсов Kubernetes» покажет, как в промышленных условиях можно применять почти любые самые актуальные ресурсы Kubernetes. Вы узнаете, как определять и доставлять их, а также управлять их версиями.

Глава 7 «Работа с хранилищем данных в Kubernetes» познакомит вас с драйверами для постоянных томов с данными и покажет, как с ними работать. Вы узнаете, как работает Flocker, и научитесь интегрировать в Kubernetes существующие хранилища данных уровня предприятия (iSCSI/NFS/FC).

Глава 8 «Запуск приложений с сохранением состояния» научит преобразовывать монолитные системы, обладающие состоянием, в микросервисы, которые работают внутри Kubernetes и способны выдерживать промышленные нагрузки. Вам будет представлено несколько способов добиться этого с ресурсом PetSet или без него в версиях Kubernetes, предшествовавших 1.3. Вы сможете заполнить пробелы, имеющиеся в действующей документации.

Глава 9 «Плавающие обновления, масштабирование и квоты» покажет, как выкатывать обновления и как ведет себя горизонтальное автомасштабирование подов. Вы научитесь выполнять и подстраивать под себя проверки масштабируемости в промышленных условиях. И сможете использовать квоты на процессорные ресурсы и память.

Глава 10 «Продвинутая организация сети в Kubernetes» позволит вам определить, какие сетевые дополнения подходят в тех или иных ситуациях, научиться развертывать их вместе с Kubernetes и расширять их возможности. Вы также овладеете балансировкой нагрузки с помощью iptables.

Глава 11 «Запуск Kubernetes в нескольких облаках и многокластерном режиме» познакомит вас с разными вариантами развертывания кластеров Kubernetes в промышленной среде. Будут представлены инструкции относительно того, как выделять, запускать и автоматизировать кластеры в Amazon и Google Cloud Engine, в том числе географически распределенные многокластерные системы с использованием Workload API.

Глава 12 «Настройка Kubernetes: API и дополнения» поможет вам реализовать и интегрировать сторонние ресурсы в имеющиеся среды и понять принцип усовершенствования API Kubernetes. В конце вы также научитесь применять внешние нестандартные механизмы распределения нагрузки при локальном развертывании с использованием популярных серверов `haproxy` или `nginx`.

Глава 13 «Поддержка диспетчера пакетов Kubernetes» покажет, как работать с приложениями Kubernetes, поставляемыми в виде пакетов. В начале этой главы вы познакомитесь с диспетчером Helm Classic, а затем перейдете к изучению Helm for Kubernetes. В конце будут приведены примеры создания и обновления пакетов в репозитории Helm, которые можно в дальнейшем развертывать и обслуживать в промышленных условиях.

Глава 14 «Будущее Kubernetes» научит вас создавать и сохранять в репозитории Helm собственные пакеты. Вы узнаете, как с помощью конвейеров доставки Kubernetes-пакеты попадают из репозитория в кластеры.

Как извлечь из книги максимальную пользу

Для выполнения примеров, которые приводятся в издании, нужно установить на компьютер последние версии Docker и Kubernetes (желательно 1.10). Система Windows 10 Professional позволяет включить режим гипервизора. Если используется другая ОС, придется установить VirtualBox, содержащий Linux.

Полноцветные изображения

PDF-файл с цветными оригинальными снимками экранов и схемами, приведенными в книге, вы можете скачать по адресу https://www.packtpub.com/sites/default/files/downloads/MasteringKubernetesSecondEdition_ColorImages.pdf.

Условные обозначения

В данной книге при форматировании текста используется ряд условных обозначений.

КодВТексте. Обозначает участки кода, имена папок и файлов, файловые расширения, пути, названия таблиц в базах данных и пользовательский ввод, например: «Проверим узлы кластера с помощью команды `get nodes`».

Блок кода выглядит следующим образом:

```
type Scheduler struct {  
    config *Config  
}
```

Любой ввод или вывод в командной строке записывается так:

```
> kubectl create -f candy.yaml  
candy "chocolate" created
```

Рубленый шрифт. Обозначает URL-адреса, текст в графическом интерфейсе и псевдонимы в Twitter.

Курсив. Обозначает новый термин или важное слово.



Предупреждения и важные замечания выглядят таким образом.



Советы и подсказки выглядят так.

1

Архитектура Kubernetes

Kubernetes — это одновременно крупный открытый проект и экосистема с большим количеством кода и богатой функциональностью. Автор Kubernetes — компания Google, но со временем этот проект присоединился к организации *Cloud Native Computing Foundation (CNCF)* и стал явным лидером в области контейнерных приложений. Если говорить коротко, это платформа для оркестрации развертывания и масштабирования контейнерных приложений и управления ими. Вы, вероятно, уже читали об этой системе и, может быть, даже применяли ее в своих проектах — любительских или профессиональных. Но, чтобы понять ее суть, научиться эффективно использовать и освоить передовой опыт, этого явно недостаточно.

В данной главе будет заложен фундамент знаний, необходимых для полноценного применения Kubernetes. Сначала я попытаюсь объяснить, чем эта система является, а чем — нет и что собой представляет оркестрация контейнеров. Затем будут рассмотрены важные концепции, которые встречаются на протяжении всей книги. После этого мы углубимся в архитектуру Kubernetes и посмотрим, каким образом пользователи могут задействовать все ее возможности. Далее обсудим различные среды выполнения и механизмы контейнеризации, которые поддерживаются в Kubernetes (Docker — лишь один из вариантов). В конце поговорим о роли Kubernetes в замкнутой непрерывной интеграции и цикле развертывания.

По прочтении этой главы вы получите четкое понимание того, что такое оркестрация контейнеров. Будете знать, какие проблемы решает Kubernetes, чем продиктованы особенности структуры и архитектуры данной системы и какие среды выполнения она поддерживает. Вы также ознакомитесь со структурой открытого репозитория и сможете самостоятельно находить ответы на любые вопросы.

Что такое Kubernetes

Kubernetes — это платформа, которая вобрала в себя огромное и непрерывно растущее количество сервисов и возможностей. Ее основная функция заключается в планировании рабочей нагрузки на контейнеры в рамках вашей инфраструктуры, и это далеко не все. Далее перечислены некоторые дополнительные возможности Kubernetes:

- ❑ мониторинг систем хранения данных;
- ❑ распространение секретной информации;
- ❑ проверка работоспособности приложений;
- ❑ репликация узлов с приложениями;

- ☐ применение горизонтального автомасштабирования подов;
- ☐ именованное и обнаружение;
- ☐ балансирование нагрузки;
- ☐ «обкатка» обновлений;
- ☐ мониторинг ресурсов;
- ☐ доступ к журнальным файлам и их обработка;
- ☐ отладка приложений;
- ☐ предоставление аутентификации и авторизации.

Чем Kubernetes не является

Kubernetes — это не *PaaS* (*platform as a service* — «платформа как услуга»). Она позволяет вам самостоятельно выбирать большинство важных функций будущей системы или предоставить этот выбор другим системам, построенным поверх Kubernetes, таким как Deis, OpenShift и Eldarion. В частности, Kubernetes:

- ☐ не диктует выбор определенного типа приложений или фреймворка;
- ☐ не требует выбирать конкретный язык программирования;
- ☐ не предоставляет базы данных или очереди сообщений;
- ☐ не делает различия между приложениями и сервисами;
- ☐ не предоставляет магазин сервисов с возможностью развертывания одним щелчком кнопкой мыши;
- ☐ дает возможность пользователям выбирать собственные системы журналирования, мониторинга и оповещения.

Оркестрация контейнеров

Основная функция Kubernetes заключается в оркестрации контейнеров, то есть планировании работы контейнеров разной степени загруженности на физических и виртуальных устройствах. Контейнеры должны быть эффективно упакованы, им необходимо соблюдать ограничения, налагаемые средой развертывания и конфигурацией кластера. Кроме того, платформа Kubernetes должна следить за всеми запущенными контейнерами и заменять те из них, которые вышли из строя, перестали отвечать или испытывают какие-то другие сложности. Kubernetes предоставляет множество других возможностей, о которых вы узнаете из следующих глав. В этом разделе мы сосредоточимся на контейнерах и оркестрации.

Контейнеры на физических и виртуальных устройствах

Все в компьютерном мире сводится к аппаратному обеспечению. Для выполнения работы вам нужно выделить реальные устройства, включая физические компьютеры с определенной вычислительной мощностью (центральными процессорами или

ядрами), памятью и каким-то локальным постоянным хранилищем данных (жесткими дисками или SSD). Потребуется также общее хранилище данных и сетевое соединение, чтобы объединить все эти компьютеры и позволить им общаться друг с другом. В наши дни, помимо использования «голого железа», вы также можете запустить несколько виртуальных машин (ВМ) на одном и том же компьютере. Кластеры, на которых можно разворачивать Kubernetes, могут быть как физическими, основанными на реальных устройствах, так и виртуальными, состоящими из виртуальных машин. Теоретически физические и виртуальные устройства комбинируются, однако такой подход не очень распространен.

Преимущества контейнеров

Контейнеры олицетворяют собой настоящий прорыв в парадигме разработки и выполнения крупных, сложных программных систем. По сравнению с традиционными моделями они имеют такие преимущества, как:

- ❑ быстрое создание и развертывание приложений;
- ❑ непрерывная разработка, интеграция и развертывание;
- ❑ разграничение ответственности разработчиков и администраторов;
- ❑ однородность сред разработки, тестирования и промышленного использования;
- ❑ переносимость между разными облачными провайдерами и ОС;
- ❑ сосредоточение управления непосредственно на приложениях;
- ❑ слабо связанные, распределенные, эластичные, независимые микросервисы;
- ❑ изоляция ресурсов;
- ❑ утилизация ресурсов.

Контейнеры в облаке

Контейнеры обеспечивают изоляцию микросервисов, при этом они сильно упрощены, что делает их идеальным средством упаковки. К тому же, в отличие от виртуальных машин, они не требуют значительных накладных расходов. Таким образом, контейнеры как нельзя лучше подходят для облачного развертывания, в условиях которого выделять каждому микросервису по виртуальной машине было бы слишком расточительно.

Все основные облачные провайдеры, такие как Amazon AWS, Google's GCE, Microsoft's Azure и даже Alibaba Cloud, предоставляют услуги по размещению контейнеров. Система GKE от Google изначально основывается на Kubernetes. AWS ECS имеет собственный механизм оркестрации. Microsoft Azure поддерживает контейнеры за счет Apache Mesos. Kubernetes можно разворачивать на любых облачных платформах, но лишь недавно интеграция этой системы с другими сервисами стала достаточно глубокой. В конце 2017 года все облачные провайдеры объявили о непосредственной поддержке Kubernetes. Компания Microsoft запустила AKS, в AWS появилась услуга EKS, а в Alibaba Cloud начались работы по прозрачной интеграции Kubernetes с помощью Kubernetes controller manager.

От мелких домашних животных к крупному рогатому скоту

В прошлом, когда системы были небольшими, у каждого сервера было имя. Разработчики в точности знали, какое программное обеспечение на каждом устройстве используется. Помнится, во многих компаниях, в которых я работал, шли многодневные споры о том, в честь чего должны называться наши серверы. Среди популярных вариантов были композиторы и герои греческих мифов. Отношение к серверам было очень нежным, мы считали их чем-то вроде любимых домашних животных. Выход сервера из строя был целым происшествием. Все собирались вместе и обсуждали: где бы взять новый сервер, что было запущено на неисправном устройстве и как перенести это на новый компьютер. Если там хранились важные данные, оставалось лишь надеяться, что у нас есть резервная копия, которая, может быть, даже подлежит восстановлению.

Очевидно, что такой подход не масштабируется. Когда количество серверов переваливает за несколько десятков и тем более сотен, вам приходится обращаться с ними как с крупным рогатым скотом. Вы должны думать обо всей стаде, а не об отдельных животных. У вас по-прежнему могут быть домашние любимцы, но веб-серверы не должны входить в их число.

Kubernetes идеально вписывается в данную концепцию, принимая на себя всю ответственность за распределение контейнеров по конкретным компьютерам. В большинстве случаев вам не нужно взаимодействовать с отдельными устройствами (узлами). Лучше всего это подходит для приложений, которые не сохраняют свое состояние. В остальных случаях все немного сложнее, однако в Kubernetes для этого предусмотрено решение под названием `StatefulSet`, которое мы вскоре обсудим.

В данном разделе мы рассмотрели идею оркестрации контейнеров, поговорили о том, как они соотносятся с серверами — физическими или виртуальными, и взвесили преимущества от выполнения контейнеров в облаке. А в конце сделали абстрактное сравнение домашних питомцев со скотом. В следующем разделе мы познакомимся с миром Kubernetes — его концепциями и терминологией.

Концепции, лежащие в основе Kubernetes

В данном разделе я вкратце познакомлю вас со многими важными концепциями, лежащими в основе Kubernetes, и попытаюсь объяснить, зачем они нужны и как связаны между собой. Это делается для того, чтобы вы смогли ориентироваться в этих концепциях и терминах. Позже вы увидите, как все сплетается воедино, помогая сформировать группы API и категории ресурсов, чтобы в итоге мы получили отличные результаты. Многие из данных концепций можно рассматривать в качестве отдельных строительных блоков. Некоторые из них, например узлы и ведущие элементы, реализованы в виде набора компонентов. Эти компоненты находятся на другом уровне абстракции, и им посвящен раздел «Ведущие компоненты Kubernetes». Знаменитая схема архитектуры Kubernetes приведена на рис. 1.1.

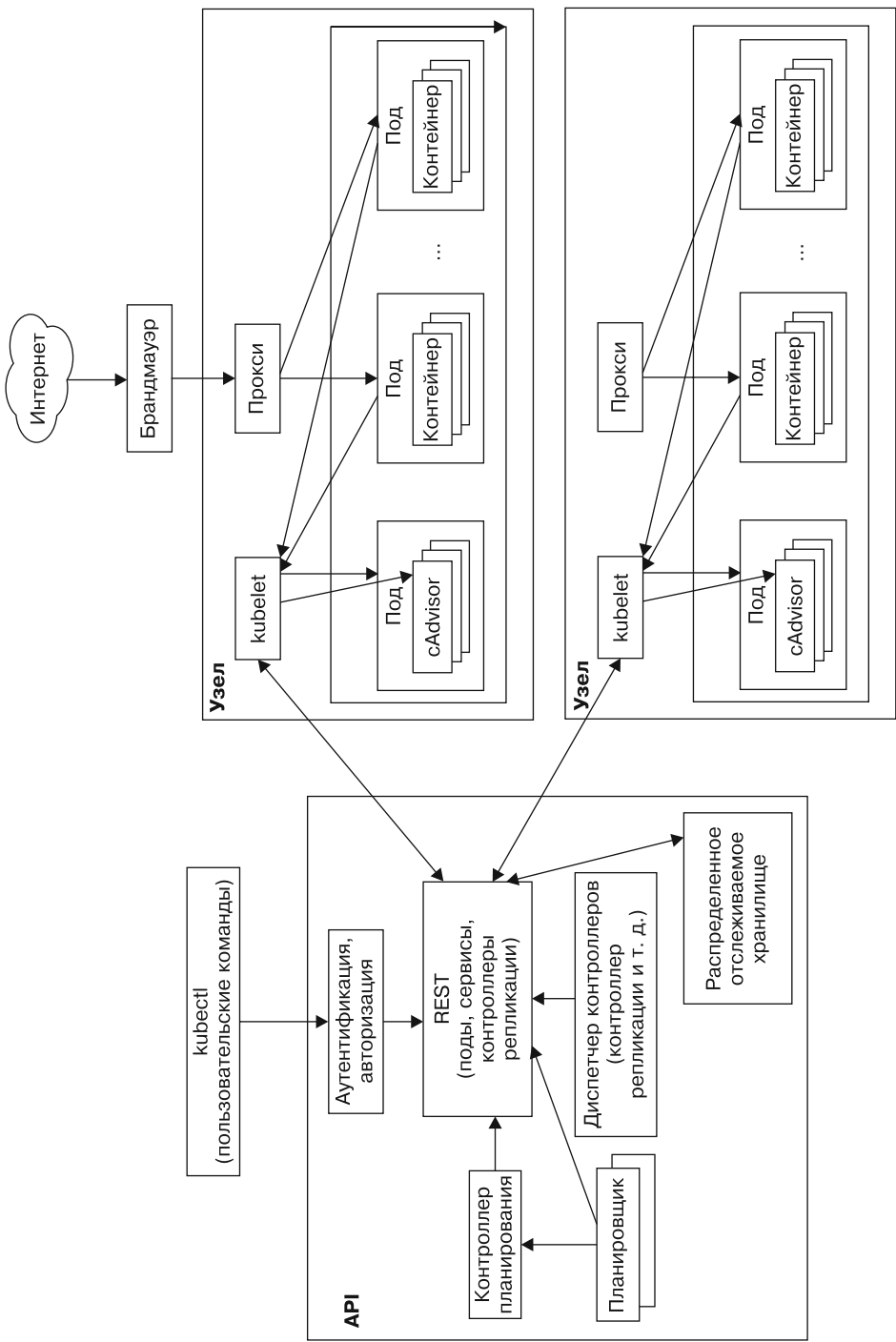


Рис. 1.1

Кластер

Кластер — это набор компьютеров, хранилищ данных и сетевых ресурсов, с помощью которых Kubernetes выполняет различные задачи в вашей системе. Стоит отметить, что система может состоять из нескольких кластеров. Позже мы подробно рассмотрим нетривиальный сценарий с многокластерным режимом.

Узел

Узел — это отдельный компьютер (физический или виртуальный). Его задача состоит в запуске подов, о которых мы поговорим чуть позже. Каждый узел в Kubernetes содержит несколько компонентов, таких как kubelet и прокси kube. Все они находятся под управлением ведущего узла. Узлы — это нечто наподобие рабочих пчел, которые занимаются выполнением всех основных задач. Раньше они назывались *миньонами* (*minions*), поэтому не удивляйтесь, если встретите этот термин в старой документации. Миньоны — это те же узлы.

Ведущий узел

Ведущий узел — это панель управления Kubernetes. Он состоит из нескольких компонентов, таких как API-сервер, планировщик и диспетчер контроллеров, и отвечает за глобальное (уровня кластера) планирование работы подов и обработку событий. Обычно все ведущие компоненты размещаются на едином узле. Но если сценарий предполагает высокую доступность или кластеры очень велики, следует подумать о том, чтобы сделать ведущий узел избыточным. Мы подробно обсудим высокодоступные кластеры в главе 4.

Под

Под (*pod*) — это единица работы в Kubernetes. Каждый под содержит один или несколько контейнеров. Поды всегда работают совместно, то есть на одном компьютере. Все контейнеры внутри пода имеют одни и те же IP-адрес и пространство портов, они могут общаться между собой через локальный сервер или посредством межпроцессного взаимодействия. Кроме того, все контейнеры имеют доступ к общему локальному хранилищу данных узла, на котором находится под. Такое хранилище может быть подключено к каждому контейнеру. Поды — важный элемент Kubernetes. В одном и том же Docker-контейнере можно запускать сразу несколько приложений, используя сервис supervisorд для управления разными процессами, однако такой подход обычно проигрывает методу с применением подов. Рассмотрим причины этого.

- ❑ **Прозрачность.** Когда контейнеры внутри пода видны извне, это позволяет предоставлять им различные инфраструктурные услуги, такие как управление процессами и мониторинг ресурсов. Таким образом пользователи получают множество удобных возможностей.

- ❑ **Разделение программных зависимостей.** Мы можем управлять версиями отдельных контейнеров, заново их перестраивать и развертывать. Возможно, в Kubernetes когда-то появится поддержка горячих обновлений отдельных контейнеров.
- ❑ **Простота использования.** Пользователям не нужно запускать собственные диспетчеры процессов, беспокоиться о передаче сигналов и кодов завершения и т. д.
- ❑ **Эффективность.** Поскольку инфраструктура берет на себя большую ответственность, контейнеры могут быть упрощенными.

Поды обеспечивают отличное решение для управления группами тесно связанных между собой контейнеров, которые для выполнения своей задачи должны взаимодействовать на одном и том же узле. Важно помнить, что поды считаются фиктивными расходными сущностями, которые при желании можно удалить или заменить. Вместе с подом уничтожается любое хранилище данных, которое в нем находилось. Каждый экземпляр пода получает *уникальный идентификатор* (*unique ID*, или *UID*), чтобы при необходимости их можно было различить.

Метка

Метка — это пара «ключ — значение», с помощью нее группируются наборы объектов (зачастую подов). Этот механизм играет важную роль для нескольких других концепций, таких как контроллеры репликации, наборы реплик и сервисы, которые работают с динамическими группами объектов и должны как-то идентифицировать членов данных групп. Между объектами и метками существует связь вида $N \times N$. Каждый объект может иметь несколько меток, и каждая метка применима к разным объектам. Метки связаны определенными ограничениями, и это сделано намеренно. Например, каждая метка объекта должна иметь уникальный ключ, соответствующий строгому синтаксису. Он должен состоять из двух частей — префикса и имени. Префикс необязателен и отделяется от имени косой чертой (/). Он должен представлять собой корректный поддомен DNS и не может содержать свыше 253 символов. Имя обязано быть, его длина ограничена 63 символами. Имена должны начинаться и заканчиваться алфавитно-цифровыми символами (a — z, A — Z, 0–9) и содержать только буквы, цифры, точки, тире и подчеркивания. Значения подчиняются тем же правилам, что и имена. Следует отметить, что метки предназначены лишь для идентификации объектов, а для внедрения произвольных метаданных используются аннотации.

Аннотации

Связывать произвольные метаданные с объектами позволяют аннотации. Kubernetes всего лишь хранит аннотации и делает доступными их метаданные. В отличие от меток они не имеют строгих ограничений относительно допустимых символов и длины.

Мой опыт свидетельствует, что такие метаданные всегда необходимы в сложных системах. И замечательно, что в Kubernetes это понимают и предоставляют их в стандартной поставке, чтобы не приходилось проектировать собственное хранилище метаданных и связывать его с объектами.

Мы охватили большинство концепций Kubernetes, осталось еще несколько, которые я уже вскользь упоминал. В следующем разделе продолжим знакомство с архитектурой Kubernetes. Мы рассмотрим мотивацию, которая стоит за теми или иными структурными решениями, изучим содержимое и реализацию данной системы и даже заглянем в ее исходный код.

Селекторы меток

Селекторы меток используются для выбора объектов на основе их меток. Тожественные селекторы указывают имя ключа и значение. Для обозначения равенства или неравенства значений предусмотрены два оператора, = (или ==) и !=, например:

```
role = webserver
```

Это позволит получить все объекты, у которых метка содержит соответствующие ключ и значение.

Селектор метки может иметь несколько условий, разделенных запятой, например:

```
role = webserver, application != foo
```

Селекторы могут основываться на наборе из нескольких значений:

```
role in (webserver, backend)
```

Контроллеры репликации и наборы реплик

Контроллеры репликации и *наборы реплик* предназначены для управления группами подов, выбранных с помощью селекторов меток. Они гарантируют, что определенное количество экземпляров подов всегда находится в рабочем состоянии. Главное различие между ними состоит в том, что контроллеры репликации проверяют принадлежность к группе по одному имени, а наборы реплик позволяют указать несколько значений. Наборы реплик считаются предпочтительным инструментом, так как контроллеры реплик — это их подмножество и, как мне кажется, в какой-то момент они будут признаны устаревшими.

Kubernetes гарантирует, что количество работающих подов всегда соответствует значению, которое вы указали в контроллере репликации или наборе реплик. Каждый раз, когда это количество уменьшается в результате проблем с узлом или самим подом, Kubernetes запускает новые экземпляры. Имейте в виду, что, если вы сами превысите указанный лимит при ручном запуске подов, контроллер репликации удалит лишние.

Раньше контроллеры репликации играли главную роль во многих сценариях, таких как выкатывание обновлений и запуск одноразовых задач. Но с разви-

тием Kubernetes многие из этих сценариев получили поддержку в виде отдельных объектов, например *Deployment*, *Job* и *DaemonSet*. Все они будут рассмотрены позже.

Сервисы

Сервисы применяются для предоставления пользователям или другим сервисам определенных функций. Обычно они состоят из группы подов, которые, как несложно догадаться, идентифицируются с помощью меток. Сервисы могут предоставлять доступ к внешним ресурсам или экземплярам подов, которыми вы управляете напрямую на уровне виртуальных IP-адресов. В Kubernetes доступ к стандартным сервисам можно получить с помощью удобных конечных точек. Следует отметить, что сервисы работают на третьем сетевом уровне (TCP/UDP). В Kubernetes 1.2 появился объект *Ingress*, который предоставляет доступ к HTTP-объектам (подробней об этом чуть позже). Существует два механизма для публикации и поиска сервисов: DNS и переменные среды. Нагрузку на сервисы способна регулировать автоматически сама система Kubernetes, но если при этом применяются внешние ресурсы или требуется особое обращение, разработчики могут выполнять балансировку вручную.

Я опустил множество технических деталей, связанных с пространствами портов и реальными/виртуальными IP-адресами. В последующих главах они будут рассмотрены во всех подробностях.

Том

Локальное хранилище в поде — элемент временный и удаляется вместе с подом. Иногда этого достаточно, если вам всего лишь нужно передавать информацию между контейнерами на одном узле. Но иногда данные должны сохраняться и после уничтожения пода или быть доступными для нескольких его экземпляров. На этот случай предусмотрена концепция *томов*. Docker тоже имеет поддержку томов, но она довольно ограничена (хотя ее постоянно развивают). Платформа Kubernetes использует собственные тома и поддерживает дополнительные типы контейнеров, такие как *rkt*, поэтому принципиально не может полагаться на аналогичный механизм в Docker.

Существует множество разновидностей томов. Многие из них напрямую поддерживаются в Kubernetes, но современный подход предполагает добавление новых типов томов через интерфейс *CSI* (*Container Storage Interface* — интерфейс хранилища контейнеров), который мы подробно обсудим чуть позже. Тома типа *emptyDir* подключаются к каждому контейнеру и основываются на содержимом материнской системы. При желании их можно разместить в памяти. Это хранилище удаляется при уничтожении пода. Существует много разных видов томов для определенных облачных платформ, сетевых файловых систем и даже репозиториях Git. В качестве любопытного примера приведу том *persistentDiskClaim*, который использует стандартное для вашей среды хранилище (обычно в облаке), инкапсулируя кое-какие детали.

StatefulSet

Поды появляются и исчезают, и если вам важны содержащиеся в них данные, можете задействовать постоянное хранилище. Это неплохой вариант, но иногда возникает необходимость в автоматическом распределенном хранилище данных, например MySQL Galera. Подобные кластеризованные системы распределяют данные по разным узлам с уникальными идентификаторами. Вы добьетесь этого с помощью обычных подов и сервисов или если воспользуетесь StatefulSet. Помните, мы сравнивали серверы с домашними животными и крупным рогатым скотом и объясняли, почему скот — правильная метафора? Так вот, StatefulSet находится где-то посередине. По аналогии с наборами реплик данная концепция гарантирует, что в любой момент у нас есть столько-то «домашних питомцев» с уникальными идентификаторами. Каждый такой «питомец» имеет:

- ❑ стабильное сетевое имя, доступное через DNS;
- ❑ порядковый номер;
- ❑ стабильное хранилище, привязанное к порядковому номеру и сетевому имени.

StatefulSet может помочь с обнаружением узлов, а также их добавлением и удалением.

Конфиденциальная информация

Конфиденциальная информация, такая как учетные данные и сведения о токенах, хранится в виде небольших объектов. Они находятся внутри хранилища etcd. Если поду нужен доступ к этим объектам, он может подключить сервер Kubernetes API в виде набора файлов с помощью выделенных секретных томов, ссылающихся на обычные тома с данными. Один и тот же секретный объект реально подключить к нескольким экземплярам пода. Платформа Kubernetes самостоятельно создает секретные объекты для своих компонентов, и вы можете делать то же самое. В качестве альтернативы конфиденциальную информацию можно размещать в переменных среды. Заметьте, что для пущей безопасности подобная информация в подах всегда находится в памяти (в `tmpfs`, если речь идет о подключении секретных объектов).

Имена

Каждый объект в Kubernetes идентифицируется по идентификатору UID и имени. Имена позволяют ссылаться на объекты в API-вызовах, могут состоять из цифр, букв в нижнем регистре, тире (-) и точек (.), а их длина не должна превышать 253 символов. Если удалить объект, вместо него можно создать новый с тем же именем, но UID должен быть уникальным в рамках жизненного цикла кластера. UID генерируются платформой Kubernetes, так что вы об этом заботиться не должны.

Пространства имен

Пространство имен — это виртуальный кластер. Один физический кластер может содержать несколько виртуальных, разделенных пространствами имен. Все виртуальные кластеры изолированы друг от друга, а общаться могут только через публичные интерфейсы. Нужно отметить, что объекты `node` и постоянные тома существуют вне пространств имен. Kubernetes может запланировать выполнение подов из разных пространств на одном и том же узле. И, соответственно, поды из разных пространств могут использовать одно и то же постоянное хранилище.

Чтобы обеспечить надлежащий доступ к ресурсам физического кластера и их корректное распределение, следует в ходе работы с пространствами имен учитывать сетевые политики и квоты на ресурсы.

Углубленное рассмотрение архитектуры Kubernetes

У платформы Kubernetes очень амбициозные цели. Она пытается упростить оркестрацию и развертывание распределенных систем, а также управление ими, предоставляя широкий набор возможностей и сервисов, которые должны работать во множестве различных сред, в том числе облачных. Вместе с тем она должна оставаться довольно простой, чтобы ею могли пользоваться обычные люди. Это сложная задача, которая в Kubernetes реализуется за счет максимально прозрачной, высокоуровневой, хорошо продуманной архитектуры, способствующей расширяемости и использованию дополнений. Многие участки Kubernetes жестко закодированы и привязаны к определенной среде, но их стараются вынести в отдельные дополнения, чтобы ядро системы было максимально универсальным и абстрактным.

В этом разделе мы разберем Kubernetes слой за слоем, словно луковицу. Сначала рассмотрим различные шаблоны проектирования распределенных систем и их поддержку в Kubernetes. Затем пройдемся по отдельным компонентам, которые составляют единую платформу, включая ее API. В конце взглянем на дерево исходного кода, чтобы лучше представить структуру самого проекта.

По прочтении этого раздела вы будете четко понимать архитектуру и особенности реализации Kubernetes, а также причины, по которым принимались те или иные проектные решения.

Шаблоны проектирования распределенных систем

Перефразируя знаменитую цитату Толстого из «Анны Карениной», можно сказать, что все счастливые (рабочие) распределенные системы похожи друг на друга. Это означает следующее: чтобы нормально работать, хорошо спроектированная система должна придерживаться проверенных принципов. Платформа Kubernetes — не просто

панель управления. Она стремится поддерживать и обеспечивать реализацию рекомендуемых методик, предоставляя разработчикам и администраторам высокоуровневые инструменты. Рассмотрим некоторые из этих шаблонов проектирования.

Шаблон проектирования «Мотоколяска»

Этот шаблон позволяет прицепить к поду с главным приложением дополнительный контейнер. Основной контейнер ничего об этом не знает и продолжает заниматься своими делами. Отличный пример — агент централизованного журналирования. Главное приложение может вести запись прямо в `stdout`, а прицепной контейнер — передавать эти данные сервису, который агрегирует их в центральном журнале, собирающем записи из всей системы. Использование этого шаблона проектирования имеет огромные преимущества по сравнению с интеграцией централизованного журналирования в главное приложение. Во-первых, приложение освобождается от всех хлопот, связанных с ведением журнала. Во-вторых, если вы захотите обновить или изменить политику централизованного журналирования или вовсе перейти на другого провайдера, достаточно будет лишь поменять и развернуть прицепной контейнер. Контейнеры с самими приложениями остаются нетронутыми, что предохраняет их от случайных поломок.

Шаблон проектирования «Представитель»

Шаблон позволяет обращаться с удаленным сервисом как с локальным, при необходимости обеспечивая соблюдение определенных правил. Представьте, к примеру, что у вас есть кластер Redis с одним ведущим узлом для записи и множеством реплик для чтения. В этом случае локальный контейнер-представитель может играть роль прокси, предоставляя главному приложению доступ к Redis. Основной контейнер просто подключается к локальному серверу `localhost:6379` (стандартный для Redis порт) — представителю, запущенному в том же поде. Данный представитель перенаправляет запросы на запись и чтение ведущему узлу Redis и, соответственно, его репликам. Как и в случае с шаблоном «Мотоколяска», главное приложение не имеет никакого понятия о происходящем. Это может серьезно помочь при тестировании с использованием локальной копии Redis. Кроме того, при изменении конфигурации кластера Redis вам нужно будет модифицировать только контейнер-представитель, главное приложение останется блаженным в своем неведении.

Шаблон проектирования «Адаптер»

Этот шаблон предназначен для стандартизации вывода из контейнера с главным приложением. Представьте себе сервис, который выкачивается поэтапно: отчеты, которые он генерирует, могут быть несовместимы с предыдущей версией. А другие сервисы и приложения, потребляющие эти отчеты, еще не обновились. В под с главным приложением можно добавить контейнер-адаптер, который станет корректировать его вывод в соответствии со старой версией, пока все потребители

не будут обновлены. Этот контейнер находится в той же локальной системе, что и главное приложение, что позволяет ему отслеживать локальные файлы и корректировать их прямо по ходу записи.

Многоузловые шаблоны проектирования

Все шаблоны, относящиеся к отдельным узлам, поддерживаются непосредственно подами Kubernetes. Многоузловые шаблоны, например, отвечающие за выбор лидера, рабочие очереди и рассылку-сборку, напрямую не поддерживаются, но в Kubernetes их вполне можно реализовать применением подов в сочетании со стандартными интерфейсами.

API Kubernetes

Чтобы понять, какие возможности обеспечивает система, нужно уделить пристальное внимание ее API. Это позволит получить общую картину того, что вы как пользователь можете делать с данной системой. Kubernetes дает доступ к нескольким группам REST API, имеющим разное назначение и разных конечных потребителей. Часть интерфейсов используется в основном другими инструментами, но есть и такие, с которыми напрямую могут взаимодействовать разработчики. Важная особенность этих API — их постоянное развитие. Чтобы поддерживать надлежащий порядок, разработчики Kubernetes стараются избежать переименования или удаления существующих объектов и полей, отдавая предпочтение расширению кода (добавлению новых объектов и полей к уже существующим). Кроме того, все конечные точки API имеют версии, часто с поддержкой форматов альфа и бета, например:

```
/api/v1
/api/v2alpha1
```

К API можно обращаться с помощью команды `kubectl cli`, через клиентские библиотеки или напрямую посредством вызова REST API. В последующих главах мы разберем развитые механизмы аутентификации и авторизации. При наличии подходящих полномочий вы получите листинги различных объектов Kubernetes, сможете просматривать, обновлять и удалять их. Рассмотрим общие аспекты API. Лучше всего это делать через API-группы. Одни из них включены по умолчанию, а другие включаются и выключаются с помощью флагов. Например, чтобы отключить группу `batch` версии V1 и включить группу `batch` версии V2 альфа, можно при запуске API-сервера установить флаг `--runtime-config`:

```
--runtime-config=batch/v1=false,batch/v2alpha=true
```

Помимо основных ресурсов, по умолчанию включены следующие:

- ☐ `DaemonSets`;
- ☐ `Deployments`;

- ❑ HorizontalPodAutoscalers;
- ❑ Ingress;
- ❑ Jobs;
- ❑ ReplicaSets.

Категории ресурсов

API делятся не только по группам, но и по функциям. Kubernetes обладает огромным API, и разобраться в нем вам поможет разбиение по категориям. Kubernetes определяет следующие категории ресурсов.

- ❑ **Workload.** Объекты для управления контейнерами и запуска их в кластерах.
- ❑ **Обнаружение и балансирование нагрузки.** Превращают объекты workload в сервисы со сбалансированной нагрузкой, доступные извне.
- ❑ **Конфигурация и хранилище.** Объекты, позволяющие инициализировать и конфигурировать приложение, а также сохранять данные, находящиеся за пределами контейнера.
- ❑ **Кластер.** Объекты, определяющие конфигурацию самого кластера, обычно их используют только операторы кластеров.
- ❑ **Метаданные.** Объекты, с помощью которых конфигурируются другие ресурсы внутри кластера, например HorizontalPodAutoscaler для масштабирования рабочей нагрузки.

В следующих разделах будут перечислены ресурсы, относящиеся к каждой из категорий, с указанием их API-группы. Я не буду называть конкретные версии, так как API быстро преодолевают путь от альфа к бета, а затем к *GA* (*general availability* — общедоступная версия), V1, V2 и т. д.

Workloads API

Workloads API содержит следующие ресурсы:

- ❑ Container — ядро;
- ❑ CronJob — пакетные задания;
- ❑ DaemonSet — приложения;
- ❑ Deployment — приложения;
- ❑ Job — пакетные задания;
- ❑ Pod — ядро;
- ❑ ReplicaSet — приложения;
- ❑ ReplicationController — ядро;
- ❑ StatefulSet — приложения.

За создание контейнеров отвечают контроллеры, которые используют поды. Под выполняет контейнеры и предоставляет зависимости, необходимые конкрет-

ной среде, такие как общие или постоянные тома для хранения данных, конфигурация и конфиденциальная информация, внедряемая в контейнер.

Далее представлено подробное описание одной из самых обыденных операций — получения списка всех подов через REST API:

GET /api/v1/pods

Она принимает различные параметры (все они опциональные):

- ❑ `pretty` — если равен `true`, данные выводятся в удобном формате;
- ❑ `labelSelector` — выражение-селектор для ограничения результатов;
- ❑ `watch` — если равен `true`, отслеживает изменения и возвращает поток событий;
- ❑ `resourceVersion` — возвращает только те события, которые произошли после заданной версии;
- ❑ `timeoutSeconds` — срок действия операций по отслеживанию и выводу списка.

Обнаружение и балансирование нагрузки

По умолчанию объекты с рабочей нагрузкой (`workload`) доступны только внутри кластера. Чтобы открыть к ним доступ извне, нужно использовать сервисы `LoadBalancer` или `NodePort`. На этапе разработки к локально доступным `workload`-объектам можно обращаться через главный API-сервер с помощью прокси. Для этого применяется команда `kubectl proxy`. Ресурсы:

- ❑ `Endpoints` — ядро;
- ❑ `Ingress` — расширения;
- ❑ `Service` — ядро.

Конфигурация и хранилище

Динамическая конфигурация без повторного развертывания — это краеугольный камень Kubernetes и неотъемлемая часть сложных распределенных приложений в вашем кластере. Ресурсы:

- ❑ `ConfigMap` — ядро;
- ❑ `Secret` — ядро;
- ❑ `PersistentVolumeClaim` — ядро;
- ❑ `StorageClass` — хранилище;
- ❑ `VolumeAttachment` — хранилище.

Метаданные

Метаданные обычно встраиваются в ресурсы, которые конфигурируют. Например, ограниченный диапазон будет частью конфигурации пода. В большинстве случаев вам не придется иметь дело с этими объектами напрямую. Существует множество ресурсов метаданных, их полный список можно найти по адресу <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.10/#-strong-metadata-strong->.

Кластер

Ресурсы из этой категории предназначены для использования операторами кластеров, а не разработчиками. Их тоже довольно много. Далее перечислены некоторые из наиболее важных:

- ❑ `Namespace` — ядро;
- ❑ `Node` — ядро;
- ❑ `PersistentVolume` — ядро;
- ❑ `ResourceQuota` — ядро;
- ❑ `ClusterRole` — `Rbac`;
- ❑ `NetworkPolicy` — сеть.

Ведущие компоненты Kubernetes

Кластер Kubernetes состоит из нескольких ведущих компонентов, которые используются для управления им, и компонентов, размещенных на каждом узле кластера. Познакомимся с ними поближе и посмотрим, как они уживаются вместе.

Ведущие компоненты обычно размещаются лишь на одном узле, однако внутри высокодоступного или очень большого кластера могут распределяться между несколькими узлами.

API-сервер

Сервер Kube API предоставляет доступ к Kubernetes REST API. Он не обладает состоянием и хранит все данные в кластере `etcd`, поэтому его несложно горизонтально масштабировать. API-сервер олицетворяет собой панель управления Kubernetes.

Etcd

`Etcd` — это высоконадежное распределенное хранилище данных. Kubernetes хранит в нем все состояние своих кластеров. В небольших временных кластерах `etcd` можно запускать в единственном экземпляре и на одном узле с ведущими компонентами. В важных системах, требующих избыточности и высокой доступности, `etcd` обычно работает в виде кластера, состоящего из трех или даже пяти узлов.

Диспетчер контроллеров Kube

Диспетчер контроллеров Kube представляет собой набор различных управляющих инструментов, упакованных в единый двоичный файл. Он содержит контроллер репликации, `pod`-контроллер, контроллер сервисов, контроллер конечных точек и т. д. Все эти инструменты отслеживают работу кластера через API и при необходимости приводят его в нужное состояние.

Диспетчер облачных контроллеров

Облачные провайдеры могут интегрироваться в платформу Kubernetes с целью управления узлами, маршрутами, сервисами и томами. Код облака взаимодействует с кодом Kubernetes, заменяя собой некоторые функции диспетчера контроллеров Kube. При работе с диспетчером облачных контроллеров флагу `--cloud-provider` диспетчера Kube следует присвоить значение `external`. Это отключит циклы управления, которые облако перехватывает. Диспетчер облачных контроллеров был представлен в Kubernetes 1.6 и уже используется несколькими облачными провайдерами.



Чтобы вам было легче воспринимать код, сделаю небольшое замечание о языке Go. Вначале идет название метода, а затем его параметры, заключенные в скобки. Каждый параметр состоит из имени и типа, записанных последовательно. В конце указываются возвращаемые значения (в языке Go их может быть несколько). Обычно, помимо самого результата, возвращают объект `error`. Если все прошло как нужно, этот объект равен `nil`.

Далее показан главный интерфейс пакета `cloudprovider`:

```
package cloudprovider
import (
    "errors"
    "fmt"
    "strings"
    "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/types"
    "k8s.io/client-go/informers"
    "k8s.io/kubernetes/pkg/controller"
)

// Это абстрактный расширяемый интерфейс для облачных провайдеров
type Interface interface {
    Initialize(clientBuilder controller.ControllerClientBuilder)
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

Большинство этих методов возвращают другие интерфейсы со своими методами внутри. Например, интерфейс `LoadBalancer`:

```
type LoadBalancer interface {
    GetLoadBalancer(clusterName string, service *v1.Service)
        (status *v1.LoadBalancerStatus, exists bool, err error)
    EnsureLoadBalancer(clusterName string, service *v1.Service, nodes []*v1.Node)
        (*v1.LoadBalancerStatus, error)
```

```

    UpdateLoadBalancer(clusterName string, service *v1.Service, nodes []*v1.Node)
    error
    EnsureLoadBalancerDeleted(clusterName string, service *v1.Service)
    error
}

```

kube-scheduler

Компонент `kube-scheduler` занимается планированием развертывания подов на узлах. Это крайне сложная задача, которая требует учета множества факторов, зависящих друг от друга, например:

- ☐ требований к ресурсам;
- ☐ требований к сервисам;
- ☐ политики аппаратных/программных ограничений;
- ☐ принадлежности и непринадлежности узлов;
- ☐ принадлежности и непринадлежности подов;
- ☐ ограничений и допусков;
- ☐ местонахождения данных;
- ☐ предельных сроков.

Если вам нужна какая-то особая логика планирования, не предусмотренная в `kube-scheduler`, можете создать собственный планировщик и запускать его самостоятельно или в сочетании с `kube-scheduler`, чтобы он управлял лишь определенным подмножеством подов.

DNS

Начиная с версии 1.3, в стандартный кластер Kubernetes входит DNS-сервер. Он размещается в обычном поде и назначает имена всем сервисам, кроме неуправляемых. Поды тоже могут иметь DNS-имена. Это крайне удобно для автоматического обнаружения.

Узловые компоненты

Узлам кластера необходимы определенные компоненты, с помощью которых они взаимодействуют с ведущими узлами, а также принимают и выполняют `workload`-объекты, информируя кластер об их состоянии.

Прокси-сервер

Прокси-сервер Kube отвечает за низкоуровневые сетевые функции на каждом узле. Он предоставляет локальный доступ к сервисам Kubernetes и может выполнять перенаправление в TCP и UDP. Для поиска IP-адресов в кластере используются переменные среды или DNS.

Kubelet

Kubelet — это представитель Kubernetes в узле. Он отвечает за взаимодействие с ведущими компонентами и управляет запущенными подами. Его обязанности:

- ❑ загрузка конфиденциальных данных пода с API-сервера;
- ❑ подключение томов;
- ❑ запуск контейнера пода (через CRI или rkt);
- ❑ уведомление о состоянии узла и каждого экземпляра пода;
- ❑ проверка работоспособности контейнеров.

В этом разделе мы погрузились в содержимое платформы Kubernetes, исследовали ее архитектуру (в общих чертах), шаблоны проектирования, которые она поддерживает, а также ее API и компоненты, используемые для управления кластером. В следующем разделе пройдемся по разным средам выполнения, совместимым с Kubernetes.

Среды выполнения, совместимые с Kubernetes

Изначально платформа Kubernetes была совместима лишь с одной средой выполнения контейнеров — Docker. Но с тех пор круг поддерживаемых сред значительно расширился:

- ❑ Docker (через слой CRI);
- ❑ Rkt (непосредственную интеграцию планируется заменить поддержкой rktlet);
- ❑ Cri-o;
- ❑ Frakti (Kubernetes в гипервизоре, ранее известная как Hypernetes);
- ❑ Rktlet (реализация CRI для rkt);
- ❑ cri-containerd.

Разработчики Kubernetes приняли принципиальное решение сделать эту платформу независимой от конкретной среды выполнения. Для этого используется технология *CRI* (*Container Runtime Interface* — интерфейс среды выполнения контейнеров).

В этом разделе мы подробно рассмотрим CRI и познакомимся с отдельными средами выполнения. Усвоив материал, вы будете способны принять обоснованное решение по выбору той или иной среды в определенных условиях. А также будете знать, когда следует перейти на другую среду выполнения или даже объединить их несколько в одной системе.

Интерфейс среды выполнения контейнеров (CRI)

CRI представляет собой gRPC API, содержащий библиотеки и спецификации/требования для сред выполнения контейнеров, которые позволяют им интегрироваться с kubelet на одном узле. В Kubernetes 1.7 вместо внутренней интеграции

с Docker появилась интеграция, основанная на CRI. Это было большое событие, которое открыло дорогу различным реализациям, использующим инновационные изменения в области контейнеров. Kubelet не нужно интегрировать с разными средами выполнения напрямую. Этот компонент может взаимодействовать с любой контейнерной средой, совместимой с CRI. Данный механизм проиллюстрирован на рис. 1.2.

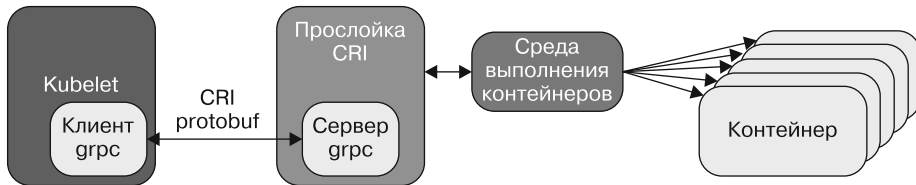


Рис. 1.2

Среды выполнения контейнеров на основе CRI (или их прослойки) должны реализовать два служебных gRPC-интерфейса: `ImageService` и `RuntimeService`. Первый отвечает за работу с изображениями. Далее показана его версия в формате protobuf (это не Go):

```
service ImageService {
    rpc ListImages(ListImagesRequest) returns (ListImagesResponse) {}
    rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResponse) {}
    rpc PullImage(PullImageRequest) returns (PullImageResponse) {}
    rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResponse) {}
    rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResponse) {}
}
```

Интерфейс `RuntimeService` управляет подами и контейнерами. Его protobuf-версия выглядит так:

```
service RuntimeService {
    rpc Version(VersionRequest) returns (VersionResponse) {}
    rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
    rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse) {}
    rpc RemovePodSandbox(RemovePodSandboxRequest) returns
        (RemovePodSandboxResponse) {}
    rpc PodSandboxStatus(PodSandboxStatusRequest) returns
        (PodSandboxStatusResponse) {}
    rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse) {}
    rpc CreateContainer(CreateContainerRequest) returns
        (CreateContainerResponse) {}
    rpc StartContainer(StartContainerRequest) returns (StartContainerResponse) {}
    rpc StopContainer(StopContainerRequest) returns (StopContainerResponse) {}
    rpc RemoveContainer(RemoveContainerRequest) returns
        (RemoveContainerResponse) {}
    rpc ListContainers(ListContainersRequest) returns (ListContainersResponse) {}
    rpc ContainerStatus(ContainerStatusRequest) returns
        (ContainerStatusResponse) {}
}
```

```

rpc UpdateContainerResources(UpdateContainerResourcesRequest) returns
    (UpdateContainerResourcesResponse) {}
rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
rpc Exec(ExecRequest) returns (ExecResponse) {}
rpc Attach(AttachRequest) returns (AttachResponse) {}
rpc PortForward(PortForwardRequest) returns (PortForwardResponse) {}
rpc ContainerStats(ContainerStatsRequest) returns (ContainerStatsResponse) {}
rpc ListContainerStats(ListContainerStatsRequest) returns
    (ListContainerStatsResponse) {}
rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns
    (UpdateRuntimeConfigResponse) {}
rpc Status(StatusRequest) returns (StatusResponse) {}
}

```

Типы данных, которые используются для аргументов и возвращаемых значений, называются *сообщениями* и тоже являются частью API. Вот как выглядит одно из таких сообщений:

```

message CreateContainerRequest {
    string pod_sandbox_id = 1;
    ContainerConfig config = 2;
    PodSandboxConfig sandbox_config = 3;
}

```

Как видите, сообщения могут встраиваться друг в друга. Одно из полей типа `CreateContainerRequest` — это строка, два других — сообщения `ContainerConfig` и `PodSandboxConfig`.

Теперь, когда вы познакомились с кодом интерфейсов Kubernetes, кратко пройдемся по отдельным средам выполнения.

Docker

Docker — настоящий тяжеловес в мире контейнеров. Изначально платформа Kubernetes предназначалась для работы исключительно с этой средой. Поддержка других сред появилась только в версии 1.3, а CRI — в 1.5. До этого в Kubernetes поддерживались лишь контейнеры Docker.

Я исхожу из того, что вы знакомы со средой Docker и ее возможностями. Эта система испытывает невиданный всплеск популярности, но вместе с тем ее довольно серьезно критикуют. Среди часто упоминаемых проблем следующие:

- ❑ недостаточная безопасность;
- ❑ сложность настройки многоконтейнерных приложений, в частности их сетевых параметров;
- ❑ недочеты в разработке, мониторинге и ведении журнала;
- ❑ ограничения, связанные с тем, что Docker-контейнеры выполняют лишь одну команду;
- ❑ слишком частый выпуск непроработанных возможностей.

Команда Docker знает о критике в свой адрес и уже исправила некоторые из этих недостатков. В частности, были сделаны инвестиции в собственную систему Docker Swarm, которая, как и Kubernetes, занимается оркестрацией. Она более проста в применении по сравнению с Kubernetes, но не такая мощная и зрелая.



Начиная с версии 1.12 демон Docker по умолчанию поддерживает режим swarm. Некоторым такое решение кажется раздутым и выходящим за рамки прямого назначения Docker, и многие в качестве альтернативы предпочитают CoreOS rkt.

С момента выпуска версии 1.11 в апреле 2016 года в платформе Docker поменялся подход к выполнению контейнеров. Теперь для контейнеризации образов формата *OCI (Open Container Initiative)* используются инструменты *containerd* и *runc* (рис. 1.3).

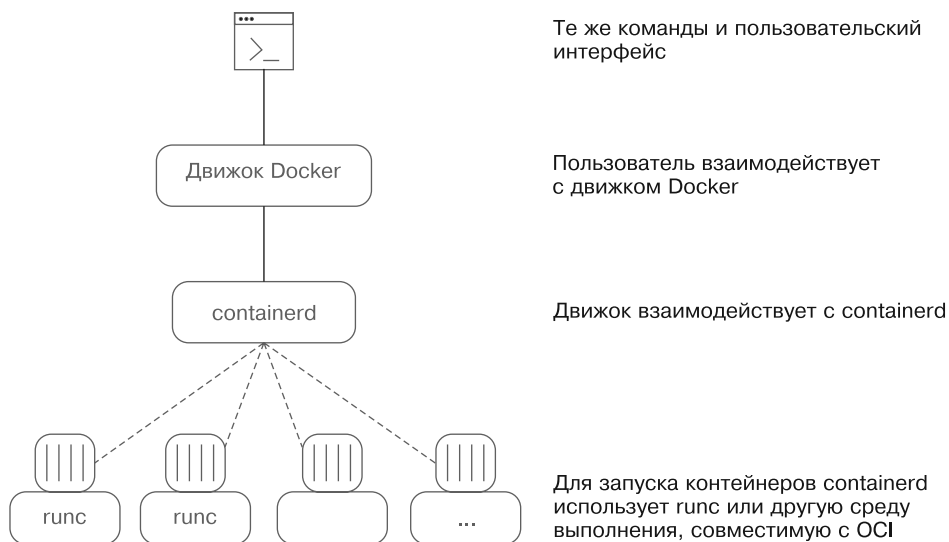


Рис. 1.3

Rkt

Rkt — это диспетчер контейнеров от команды CoreOS (на счету которой дистрибутив Linux CoreOS, etcd, flannel и т. д.). Среда выполнения rkt отличается своей простотой и делает акцент на безопасности и изоляции. У нее нет демона, как у движка Docker, для запуска своего исполняемого файла она задействует систему инициализации операционной системы — *systemd*. Rkt может загружать, проверять и запускать внутри контейнеров образы форматов *appc (app container)* и *OCI*. Архитектура этой системы куда проще, чем ее аналогов.

App container

В декабре 2014 года команда CoreOS начала разработку спецификации *appc*, которая включает в себя стандартный формат образа (ACI), среду выполнения, цифровую подпись и обнаружение. Несколькими месяцами позже разработчики Docker выступили с аналогичной инициативой относительно спецификации OCI. На сегодняшний день все идет к тому, что эти два проекта станут совместимыми друг с другом. Таким образом, данные инструменты, образы и среды выполнения смогут взаимодействовать между собой, и это прекрасно.

Cri-O

Проект Cri-O вырос из инкубатора Kubernetes. Он создавался как способ интеграции между Kubernetes и средами выполнения, совместимыми с OCI, такими как Docker. Идея состоит в том, что Cri-O будет предоставлять следующие возможности:

- ❑ поддержку разных форматов образов, включая существующий формат Docker;
- ❑ поддержку разных средств загрузки образов, включая механизмы проверки подлинности;
- ❑ управление образами контейнеров (их слоями, общими файловыми системами и т. д.);
- ❑ управление жизненным циклом контейнерных процессов;
- ❑ мониторинг и журналирование, необходимые для удовлетворения требований CRI;
- ❑ изоляцию ресурсов, отвечающую требованиям CRI.

Это позволит интегрировать в Kubernetes любую среду выполнения контейнеров, совместимую с OCI.

Rktnetes

Rktnetes — это Kubernetes плюс rkt в качестве среды выполнения. Kubernetes все еще абстрагируется от движков для выполнения контейнеров. На самом деле Rktnetes — это не отдельный продукт, того же эффекта можно добиться добавлением нескольких ключей командной строки при запуске kubelet на каждом узле.

Готова ли среда rkt к промышленному использованию?

Мой опыт работы с rkt невелик. Но эта система применяется в Tectonic — коммерческом дистрибутиве Kubernetes, основанном на CoreOS. Если вы задействуете какой-то другой тип кластера, я бы посоветовал вам подождать, пока в Kubernetes не появится интеграция с rkt через интерфейсы CRI/rktlet. При выборе rkt вместо Docker могут возникнуть определенные проблемы, например: недостающие тома не создаются автоматически, не работают команды `kubect1` для привязки и получения журнальных файлов, не поддерживаются `init`-контейнеры и т. д.

Контейнеры Hyper

Еще один вариант — контейнеры Hyper. Они содержат упрощенную виртуальную машину с собственным гостевым ядром и выполняются на «голом железе». Для изоляции вместо Linux cgroups используется гипервизор. Это интересный подход по сравнению со стандартными аппаратными кластерами, которые сложно настраивать, и публичными облаками, в которых контейнеры развертываются в тяжеловесных виртуальных машинах.

Stackube. Stackube (ранее известный как Hypernetes) — это мультиарендный дистрибутив на основе контейнеров Hyper и некоторых компонентов из OpenStack, таких как постоянное хранилище и сеть. Поскольку контейнеры не используют общее ядро узла, они могут принадлежать разным пользователям и при этом безопасно работать на одном и том же физическом компьютере. Естественно, в качестве среды выполнения в Stackube берется Frakti.

В этом разделе я познакомил вас с различными средами выполнения, которые поддерживаются в Kubernetes, не забыв упомянуть о движении в сторону стандартизации и совместимости. Дальше сделаем шаг назад и взглянем на общую картину, а затем посмотрим, как Kubernetes вписывается в концепцию непрерывной интеграции и разработки.

Непрерывная интеграция и разработка

Платформа Kubernetes отлично подходит для выполнения приложений с микросервисной архитектурой. Но, по сути, это лишь один из элементов общего процесса. Пользователи и многие разработчики могут даже не догадываться о том, где именно развертывается их система. Тем не менее Kubernetes может сделать возможными вещи, которые раньше казались слишком сложными.

В этом разделе мы рассмотрим принцип действия *непрерывной интеграции/разработки* (continuous integration and deployment, CI/CD) и то, какую роль здесь играет Kubernetes. В результате вы научитесь создавать цепочки CI/CD, используя такие преимущества Kubernetes, как простота масштабирования и применение одной и той же среды при разработке и в промышленных условиях. Это позволит сделать процесс создания и развертывания систем более продуктивным и надежным.

Цепочка CI/CD

Цепочка CI/CD — это последовательность шагов, выполняемых разработчиками или операторами по изменению кода/данных/конфигурации системы, тестированию и развертыванию ее для реального применения. Одни цепочки полностью автоматизированы, а другие требуют присмотра со стороны человека. В крупных организациях изменения могут автоматически развертываться в многоуровневых тестовых средах, нуждаясь в ручной проверке перед окончательным выпуском системы. Стандартная цепочка представлена на схеме (рис. 1.4).

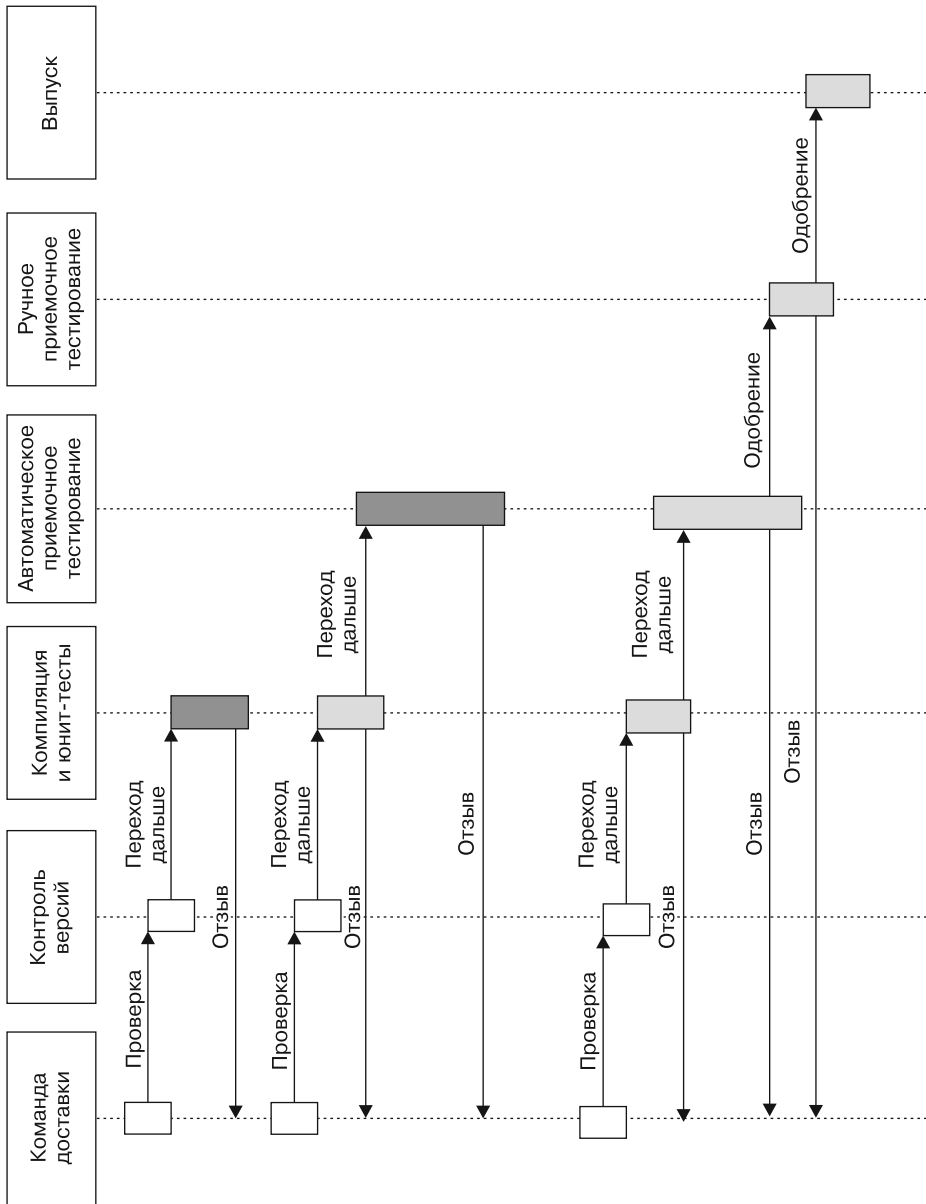


Рис. 1.4

Наверное, стоит упомянуть, что разработчики могут быть полностью изолированы от промышленной инфраструктуры и работать лишь с Git. Хороший пример такого подхода — платформа Deis (PaaS на основе Kubernetes, похожая на Heroku).

Проектирование цепочки CI/CD для Kubernetes

Развертывание на кластере Kubernetes требует пересмотра некоторых традиционных подходов. Для начала, отличается процесс упаковывания. Вам нужно подготавливать образы для своих контейнеров. Изменения, внесенные в код, можно отменить максимально просто и мгновенно благодаря умным меткам. Это дает уверенность в том, что, если плохой код каким-то образом проскользнет через мириады тестов, вы тут же сможете откатиться к предыдущей версии. Но здесь нужно быть осторожными. Миграция данных и восстановление их структуры не выполняются автоматически.

Еще одна уникальная особенность Kubernetes — то, что разработчики могут запустить локально целый кластер. На этапе проектирования это требует определенных усилий, но, поскольку микросервисы, из которых состоит система, размещаются внутри контейнеров, взаимодействующих между собой через API, эта задача становится вполне посильной и реализуемой на практике. Если же система активно работает с данными, вам придется это учитывать, создавая снимки хранилищ и предоставляя своим разработчикам синтетические наборы информации.

Резюме

В этой главе мы затронули много тем, связанных со структурой и архитектурой Kubernetes. Kubernetes — это платформа для оркестрации приложений, основанных на микросервисах и выполняющихся в виде контейнеров. Ее кластеры состоят из ведущих и рабочих узлов. Контейнеры работают внутри подов, каждый из которых размещается на физическом или виртуальном устройстве. Kubernetes имеет встроенную поддержку множества объектов, таких как сервисы, метки и постоянные хранилища. На этой платформе можно задействовать различные шаблоны проектирования распределенных систем. Контейнерные среды выполнения должны реализовывать интерфейс CRI. В число поддерживаемых входят контейнеры Docker, rkt, Nureg и др.

В главе 2 мы рассмотрим разные способы создания кластеров Kubernetes, обсудим необходимость применения тех или иных параметров и построим много-узловой кластер.

2

Создание кластеров Kubernetes

Из предыдущей главы вы узнали, что представляет собой платформа Kubernetes, как она спроектирована, какие концепции и среды выполнения поддерживает и как вписывается в процесс CI/CD.

Создание кластера Kubernetes — нетривиальная задача. Вам придется выбирать из длинного списка параметров и инструментов, а также учитывать множество факторов. В этой главе мы засучим рукава и скомпонуем несколько кластеров. Обсудим различные инструменты, такие как Minikube, kubectl, kube-spray, bootkube и stackube, и дадим им оценку. Мы рассмотрим разные среды развертывания — локальные, облачные и аппаратные. Далее приведены темы, которые мы затронем.

- ❑ Создание одноузлового кластера с помощью Minikube.
- ❑ Создание многоузлового кластера с помощью kubectl.
- ❑ Создание кластера в облаке.
- ❑ Создание кластеров с нуля на «голом железе».
- ❑ Обзор других вариантов создания кластеров Kubernetes.

Прочитав эту главу, вы будете разбираться в различных способах создания кластеров Kubernetes и в том, какие инструменты лучше всего для этого подходят. А также получите опыт компоновки кластеров — как одно-, так и многоузловых.

Быстрое создание одноузлового кластера с помощью Minikube

В этом разделе мы создадим одноузловой кластер в Windows. Выбор операционной системы обусловлен тем, что Minikube и кластеры этого типа лучше всего подходят для локальной разработки. В промышленных условиях Kubernetes обычно развертывается в Linux, но многие программисты используют компьютеры с Windows и macOS. К тому же процесс установки Minikube в Linux мало чем отличается от того, который здесь рассматривается.

Подготовка

Прежде чем создавать кластер, необходимо установить некоторые инструменты: VirtualBox, kubectl (интерфейс командной строки для Kubernetes) и, конечно же, саму утилиту Minikube. Вот их последние версии на момент написания книги:

- ❑ *VirtualBox* — www.virtualbox.org/wiki/Downloads;
- ❑ *Kubectl* — kubernetes.io/docs/tasks/tools/install-kubectl/;
- ❑ *Minikube* — kubernetes.io/docs/tasks/tools/install-minikube/.

В Windows

Установите VirtualBox и убедитесь в том, что kubectl и Minikube указаны в списке системных путей. Я просто сбрасываю все консольные программы в `c:\windows`, но у вас может быть свой подход. Я использую ConEMU для управления несколькими консолями, терминалами и сеансами SSH. Эта оболочка работает с `cmd.exe`, PowerShell, PuTTY, Cygwin, `msys` и `Git-Bash`. В Windows сложно найти что-то лучшее.



В Windows 10 Pro есть возможность использовать гипервизор Hyper-V. Формально это предпочтительный вариант, но он прочно привязан к профессиональной версии Windows. С VirtualBox все инструкции универсальны и могут применяться в разных версиях Windows и даже других операционных системах. Гипервизор Hyper-V не способен сосуществовать с VirtualBox, поэтому должен быть отключен.

Рекомендую задействовать PowerShell в режиме администратора. Вы можете добавить в свой профиль PowerShell следующие псевдоним и функцию:

```
Set-Alias -Name k -Value kubectl
function mk
{
    minikube-windows-amd64 '
    --show-libmachine-logs '
    --alsologtostderr      '
    @args
}
```

В macOS

Добавьте псевдонимы в свой файл `.bashrc` (по аналогии с псевдонимом и функцией для PowerShell в Windows):

```
alias k='kubectl'
alias mk='/usr/local/bin/minikube'
```

Теперь можете использовать сокращенные команды `k` и `mk`. Флаги Minikube, указанные в функции `mk`, обеспечивают улучшенное ведение журнала и прямой вывод в консоль и файлы аналогично команде `tee`.

Введите `mk version`, чтобы проверить корректность установки и работоспособность Minikube:

```
> mk version
```

```
minikube version: v0.26.0
```

Проверьте корректность установки и работоспособность утилиты `kubectl` с помощью команды `k version`:

```
> k version
```

```
Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.0",
GitCommit:"925c127ec6b946659ad0fd596fa959be43f0cc05", GitTreeState:"clean",
BuildDate:"2017-12-16T03:15:38Z", GoVersion:"go1.9.2", Compiler:"gc",
Platform:"darwin/amd64"}
Unable to connect to the server: dial tcp 192.168.99.100:8443: getsockopt:
operation timed out
```

Не обращайте внимания на ошибку в последней строке. Наш кластер еще не запущен, поэтому `kubectl` не может к нему подключиться. Так и должно быть.

Самостоятельно исследуйте любые команды и флаги, доступные для Minikube и `kubectl`. Здесь мы рассмотрим только те, которые используются в данном примере.

Создание кластера

Утилита Minikube поддерживает разные версии Kubernetes. На момент написания этой книги их список выглядит так:

```
> mk get-k8s-versions
```

```
The following Kubernetes versions are available when using the localkube
bootstrapper:
```

- v1.10.0
- v1.9.4
- v1.9.0
- v1.8.0
- v1.7.5
- v1.7.4
- v1.7.3
- v1.7.2
- v1.7.0
- v1.7.0-rc.1
- v1.7.0-alpha.2
- v1.6.4
- v1.6.3
- v1.6.0

- v1.6.0-rc.1
- v1.6.0-beta.4
- v1.6.0-beta.3
- v1.6.0-beta.2
- v1.6.0-alpha.1
- v1.6.0-alpha.0
- v1.5.3
- v1.5.2
- v1.5.1
- v1.4.5
- v1.4.3
- v1.4.2
- v1.4.1
- v1.4.0
- v1.3.7
- v1.3.6
- v1.3.5
- v1.3.4
- v1.3.3
- v1.3.0

Создадим кластер с помощью команды `start` и укажем последнюю стабильную версию — `v1.10.0`.

Это может занять некоторое время, так как утилита `Minikube` должна загрузить образ и настроить локальный кластер. Просто подождите, пока она закончит работу. Вы должны увидеть следующий вывод (в `Mac`):

```
> mk start --kubernetes-version="v1.10.0"
Starting local Kubernetes v1.10.0 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Finished Downloading kubeadm v1.10.0
Finished Downloading kubelet v1.10.0
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

Рассмотрим его, чтобы понять принцип работы `Minikube`. При создании кластера с нуля многие из следующих шагов придется выполнить вручную.

1. Запуск виртуальной машины `VirtualBox`.
2. Создание сертификатов для локальной системы и `VM`.
3. Загрузка образов.
4. Установление соединения между локальной системой и `VM`.
5. Запуск локального кластера `Kubernetes` внутри `VM`.

6. Настройка кластера.
7. Запуск всех компонентов панели управления Kubernetes.
8. Настройка утилиты `kubectl`, чтобы она могла общаться с кластером.

Отладка

Если во время этой процедуры что-то пойдет не так, попытайтесь разобраться в сообщениях об ошибках. Вы можете добавить флаг `--alsologtostderr`, чтобы сделать их более подробными. Все результаты работы Minikube аккуратно помещаются в каталог `~/minikube`, структура которой представлена далее:

```
> tree ~/.minikube -L 2
/Users/gigi.sayfan/.minikube
```

```
├── addons
├── apiserver.crt
├── apiserver.key
├── ca.crt
├── ca.key
├── ca.pem
├── cache
├── images
├── iso
├── localkube
├── cert.pem
├── certs
├── ca-key.pem
├── ca.pem
├── cert.pem
├── key.pem
├── client.crt
├── client.key
├── config
├── config.json
├── files
├── key.pem
├── last_update_check
├── logs
├── machines
├── minikube
├── server-key.pem
├── server.pem
├── profiles
├── minikube
├── proxy-client-ca.crt
├── proxy-client-ca.key
├── proxy-client.crt
├── proxy-client.key
```

```
13 directories, 21 files
```


и панель управления). Пришло время запустить какие-нибудь экземпляры пода. Возьмем в качестве примера сервер echo:

```
k run echo --image=gcr.io/google_containers/echoserver:1.8 --port=8080
deployment "echo" created
```

Платформа Kubernetes развернула и запустила под. Обратите внимание на префикс echo:

```
> k get pods
NAME                                READY    STATUS    RESTARTS    AGE
echo-69f7cfb5bb-wqgkh              1/1      Running   0            18s
```

Для того чтобы превратить под в сервис, доступный извне, введите следующую команду:

```
> k expose deployment echo --type=NodePort
service "echo" exposed
```

Мы выбрали тип сервиса NodePort, чтобы он был доступен локально на заданном порте (имейте в виду, что порт 8080 уже занят подом). Порты назначаются в кластере. Чтобы обратиться к сервису, нужны его IP-адрес и порт:

```
> mk ip
192.168.99.101
> k get service echo --output='jsonpath="{.spec.ports[0].nodePort}"'
30388
```

При обращении сервис echo возвращает множество информации:

```
> curl http://192.168.99.101:30388/hi
```

Поздравляю! Вы только что создали локальный кластер Kubernetes и развернули в нем сервис.

Исследование кластера с помощью панели управления

Kubernetes имеет очень приятный веб-интерфейс, который, естественно, развивается в виде сервиса внутри пода. Панель управления хорошо продумана и предоставляет общие сведения о вашем кластере, а также подробности об отдельных ресурсах. С ее помощью можно просматривать журнал, редактировать файлы ресурсов и многое другое. Это идеальный инструмент для ручной проверки кластера. Чтобы запустить панель управления, введите команду `minikube dashboard`.

Minikube откроет окно браузера с интерфейсом панели управления. Стоит отметить, что Microsoft Edge в Windows не может вывести ее. Мне пришлось воспользоваться другим браузером.

Вот как выглядит обзор workload-объектов: Deployments (Развертывание), Replica Sets (Наборы реплик), Replication Controllers (Контроллеры репликации) и Pods (Поды) (рис. 2.1).

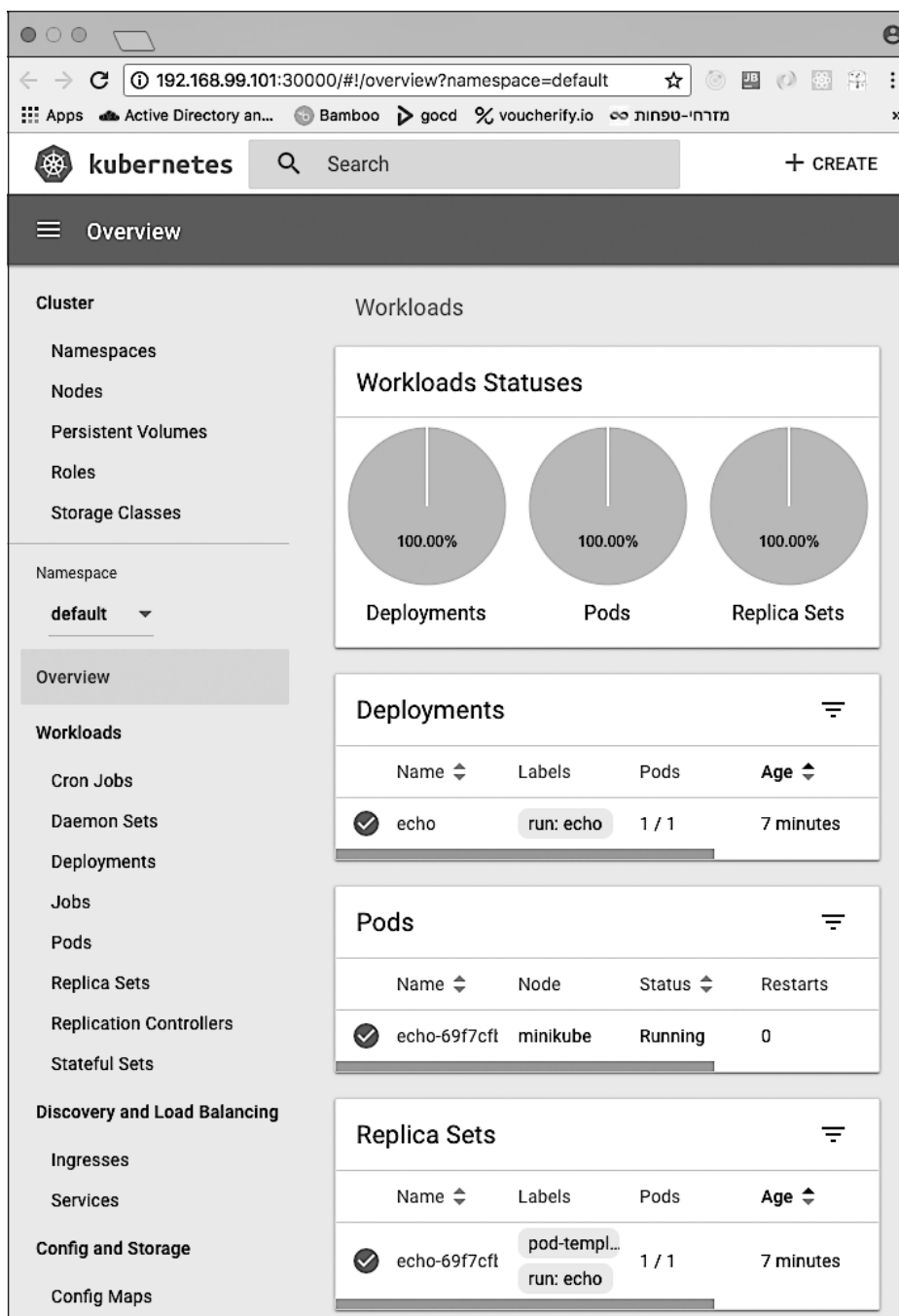


Рис. 2.1

Здесь также можно было бы просмотреть разделы **Daemon Sets** (Наборы демонов), **Stateful Sets** (Наборы с состоянием) и **Jobs** (Задания), но в этом кластере их нет.

В данном разделе мы создали локальный одноузловой кластер в Windows, немало исследовали его с помощью утилиты **kubect1**, развернули сервис и поиграли с веб-интерфейсом. Далее перейдем к многоузловым кластерам.

Создание многоузлового кластера с помощью kubernetes

В этом разделе вы познакомитесь с **kubernetes** — инструментом для создания кластеров Kubernetes, который рекомендуется к использованию во всех средах, несмотря на то что находится на стадии активной разработки. Являясь частью Kubernetes, он неизбежно вбирает в себя лучшие методики. Чтобы сделать его доступным для всего кластера, воспользуемся виртуальными машинами. Этот раздел предназначен для читателей, которые хотят получить практический опыт развертывания многоузлового кластера.

Чего следует ожидать

Прежде чем идти дальше, я хочу акцентировать ваше внимание на том, что наше путешествие может оказаться не самым гладким. Перед утилитой **kubernetes** стоит непростая задача: она должна постоянно равняться на платформу Kubernetes, которая сама не стоит на месте. Это может привести к некоторой нестабильности. Чтобы довести до удобоваримого состояния первое издание книги, мне приходилось погружаться в дебри Kubernetes и искать обходные пути. И знаете что? Со вторым изданием повторилась та же история. Поэтому приготовьтесь к тому, что вам нужно будет вносить некоторые корректировки и спрашивать совета у других. Если вы предпочитаете менее проблемные решения, позже я предложу несколько хороших вариантов.

Подготовка к работе

kubernetes работает на уже готовом устройстве (физическом или виртуальном). Перед созданием кластера Kubernetes нужно подготовить несколько ВМ и установить базовое программное обеспечение, такое как **Docker**, **kubelet**, **kubernetes** и **kubect1** (последняя утилита нужна только на ведущем узле).

Подготовка кластера виртуальных машин на основе vagrant

Следующий **vagrant**-файл создаст кластер из четырех ВМ с именами **n1**, **n2**, **n3** и **n4**. Чтобы поднять кластер, введите команду **vagrant up**. ВМ основаны на **Bento/Ubuntu**

версии 16.04, у дистрибутива Ubuntu/Xenial возникают различные проблемы, поэтому мы его здесь не используем.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
hosts = {
  "n1" => "192.168.77.10",
  "n2" => "192.168.77.11",
  "n3" => "192.168.77.12",
  "n4" => "192.168.77.13"
}
Vagrant.configure("2") do |config|
  # всегда используем небезопасный ключ Vagrant
  config.ssh.insert_key = false
  # перенаправляем ssh-агент, чтобы получить легкий доступ к разным узлам
  config.ssh.forward_agent = true

  check_guest_additions = false
  functional_vboxsf = false

  config.vm.box = "bento/ubuntu-16.04"
  hosts.each do |name, ip|
    config.vm.hostname = name
    config.vm.define name do |machine|
      machine.vm.network :private_network, ip: ip
      machine.vm.provider "virtualbox" do |v|
        v.name = name
      end
    end
  end
end
```

Установка необходимого программного обеспечения

Я очень люблю задействовать утилиту Ansible для управления конфигурациями, поэтому установил ее на ВМ n4 (под управлением Ubuntu 16.04). С этого момента n4 становится управляющим устройством — это означает, что мы будем работать в среде Linux. Я мог бы использовать Ansible прямо на своем Mac-компьютере, но, поскольку этот инструмент несовместим с Windows, сделал выбор в пользу более универсального подхода:

```
> vagrant ssh n4
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-87-generic x86_64)
* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage
0 packages can be updated.
0 updates are security updates.
```

```
vagrant@vagrant:~$ sudo apt-get -y --fix-missing install python-pip
sshpass
vagrant@vagrant:~$ sudo pip install ansible
```

Я использую версию 2.5.0, но самая последняя тоже должна подойти:

```
vagrant@vagrant:~$ ansible --version
ansible 2.5.0
  config file = None
  configured module search path =
[u'/home/vagrant/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /home/vagrant/.local/lib/python2.7/
sitepackages/ansible
  executable location = /home/vagrant/.local/bin/ansible
  python version = 2.7.12 (default, Dec 4 2017, 14:50:18)
    [GCC 5.4.0 20160609]
  python version = 2.7.12 (default, Dec 4 2017, 14:50:18)
    [GCC 5.4.0 20160609]
```

Программа `sshpass`, которую я установил, поможет `ansible` подключиться ко всем ВМ со встроенной учетной записью `vagrant`. Это имеет смысл только для локального многоузлового кластера на основе ВМ.

Я создал каталог `ansible` и поместил в него три файла: `hosts`, `vars.yml` и `playbook.yml`.

Файл host

`host` — это служебный файл, благодаря которому каталог `ansible` знает, на какие узлы он должен быть скопирован. Узлы должны быть доступны из управляющей системы по SSH. Далее перечислены три ВМ, на которые будет установлен кластер:

```
[all]
192.168.77.10 ansible_user=vagrant ansible_ssh_pass=vagrant
192.168.77.11 ansible_user=vagrant ansible_ssh_pass=vagrant
192.168.77.12 ansible_user=vagrant ansible_ssh_pass=vagrant
```

Файл vars.yml

Файл `vars.yml` всего лишь хранит список всех пакетов, которые я хочу установить на всех узлах. На каждом компьютере, который находится под моим управлением, я предпочитаю иметь `vim`, `htop` и `tmux`. Остальные пакеты требуются для Kubernetes:

```
---
PACKAGES:
- vim - htop - tmux - docker.io
- kubelet
- kubeadm
- kubectl
- kubernetes-cni
```

Файл `playbook.yml`

Запуск файла `playbook.yml` приводит к установке пакетов на все узлы:

```
---
- hosts: all
  become: true
  vars_files:
    - vars.yml
  strategy: free
  tasks:
    - name: hack to resolve Problem with MergeList Issue
      shell: 'find /var/lib/apt/lists -maxdepth 1 -type f -exec rm -v {} \;';
    - name: update apt cache directly (apt module not reliable)
      shell: 'apt-get clean && apt-get update'
    - name: Preliminary installation
      apt: name=apt-transport-https force=yes
    - name: Add the Google signing key
      apt_key: url=https://packages.cloud.google.com/apt/doc/apt-key.gpg
               state=present
    - name: Add the k8s APT repo
      apt_repository: repo='deb http://apt.kubernetes.io/ kubernetes-xenial
                        main' state=present
    - name: update apt cache directly (apt module not reliable) shell:
      'apt-get update'
    - name: Install packages
      apt: name={{ item }} state=installed force=yes
           with_items: "{{ PACKAGES }}"
```

Поскольку некоторые пакеты берутся из АРТ-репозитория Kubernetes, его тоже следует указать (вместе с цифровым ключом Google).

Подключимся к n4:

> vagrant ssh n4

Вам нужно будет зайти на каждый из узлов, задав `ssh`:

```
vagrant@vagrant:~$ ssh 192.168.77.10
vagrant@vagrant:~$ ssh 192.168.77.11
vagrant@vagrant:~$ ssh 192.168.77.12
```

Для того чтобы не выполнять эту процедуру каждый раз, можно добавить файл `~/ansible.cfg` со следующим содержанием:

```
[defaults]
host_key_checking = False
```

Запустите `playbook.yml` на узле n4 с помощью такой команды:

```
vagrant@n4:~$ ansible-playbook -i hosts playbook.yml
```



Если вам не удастся установить соединение, попробуйте еще раз. Иногда АРТ-репозиторий Kubernetes отвечает медленно. Это действие нужно проделать лишь один раз для каждого узла.

Создание кластера

Пришло время создать сам кластер. Инициализируем ведущий узел на первой ВМ, затем настроим сеть и добавим остальные ВМ в качестве узлов.

Инициализация ведущего узла. Сделаем n1 (192.168.77.10) ведущим узлом. Работая с облаком, основанным на ВМ vagrant, нужно обязательно указать флаг `--apiserveradvertise-address`:

```
> vagrant ssh n1
vagrant@n1:~$ sudo kubeadm init --apiserver-advertise-address 192.168.77.10
```

В Kubernetes 1.10.1 это вызовет появление такого сообщения об ошибке:

```
[init] Using Kubernetes version: v1.10.1
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks.
[WARNING FileExisting-crictl]: crictl not found in system path
[preflight] Some fatal errors occurred:
[ERROR Swap]: running with swap on is not supported. Please disable
swap
[preflight] If you know what you are doing, you can make a check non-fatal
with '--ignore-preflight-errors=...'
```

Дело в том, что по умолчанию не установлены инструменты из пакета cri-tools. Здесь мы имеем дело с одним из передовых аспектов Kubernetes. Я создал дополнительный раздел в файле `playbook.yml`, чтобы установить Go и cri-tools, отключить раздел подкачки и исправить сетевые имена виртуальных машин vagrant:

```
---
- hosts: all
  become: true
  strategy: free
  tasks:
    - name: Add the longsleep repo for recent golang version
      apt_repository: repo='ppa:longsleep/golang-backports' state=present
    - name: update apt cache directly (apt module not reliable)
      shell: 'apt-get update'
      args:
        warn: False
    - name: Install Go
      apt: name=golang-go state=present force=yes
    - name: Install crictl
      shell: 'go get github.com/kubernetes-incubator/cri-tools/cmd/crictl'
      become_user: vagrant
    - name: Create symlink in /usr/local/bin for crictl
      file:
        src: /home/vagrant/go/bin/crictl
        dest: /usr/local/bin/crictl
        state: link
    - name: Set hostname properly
```

```

    shell: "hostname n$((1 + $(ifconfig | grep 192.168 | awk '{print $2}' |
    tail -c 2)))"
- name: Turn off swap
  shell: 'swapoff -a'
-

```

Не забудьте повторно запустить этот файл на узле n4, чтобы обновить все ВМ в кластере.

Далее приводится часть вывода при успешном запуске Kubernetes:

```

vagrant@n1:~$ sudo kubeadm init --apiserver-advertise-address 192.168.77.10
[init] Using Kubernetes version: v1.10.1
[init] Using Authorization modes: [Node RBAC]
[certificates] Generated ca certificate and key.
[certificates] Generated apiserver certificate and key.
[certificates] Valid certificates and keys now exist in
"/etc/kubernetes/pki"
.
.
.
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy
Your Kubernetes master has initialized successfully!

```

Позже, при подключении к кластеру других узлов, вам нужно будет записать куда больше информации. Чтобы начать применять кластер, запустите от имени обычного пользователя следующие команды:

```

vagrant@n1:~$ mkdir -p $HOME/.kube
vagrant@n1:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
vagrant@n1:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config

```

Теперь, чтобы подключить к кластеру любое количество ВМ, на каждом из его узлов достаточно выполнить от имени администратора лишь одну команду. Чтобы ее получить, введите `kubeadm init command:sudo kubeadm join -token << token>> --discovery-token-ca-cert-hash <<discovery token>> -skip-prflight-checks`.

Настройка pod-сети

Сеть — это важная составляющая кластера. Подам нужно как-то общаться друг с другом. Для этого следует установить дополнение с поддержкой pod-сети. В вашем распоряжении есть несколько вариантов, однако они должны быть основаны на CNI, так как этого требуют кластеры, сгенерированные с помощью команды `kubeadm`. Я предпочитаю дополнение Weave Net, которое поддерживает ресурс Network Policy, но вы можете выбрать нечто другое.

Выполните в ведущей ВМ команду:

```

vagrant@n1:~$ sudo sysctl net.bridge.bridge-nf-call-iptables=1
net.bridge.bridge-nf-call-iptables = 1vagrant@n1:~$ kubectl apply -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 |
tr -d '\n')")

```

Вы должны увидеть следующее:

```
serviceaccount "weave-net" created
clusterrole.rbac.authorization.k8s.io "weave-net" created
clusterrolebinding.rbac.authorization.k8s.io "weave-net" created
role.rbac.authorization.k8s.io "weave-net" created
rolebinding.rbac.authorization.k8s.io "weave-net" created
daemonset.extensions "weave-net" created
```

Успешность выполнения можно проверить так:

```
vagrant@n1:~$ kubectl get po --all-namespaces
NAMESPACE NAME READY STATUS RESTARTS AGE
kube-system etcd-n1 1/1 Running 0 2m
kube-system kube-apiserver-n1 1/1 Running 0 2m
kube-system kube-controller-manager-n1 1/1 Running 0 2m
kube-system kube-dns-86f4d74b45-jqctg 3/3 Running 0 3m
kube-system kube-proxy-154s9 1/1 Running 0 3m
kube-system kube-scheduler-n1 1/1 Running 0 2m
kube-system weave-net-fl7wn 2/2 Running 0 31s
```

Последним стоит под `weave-net-fl7wn`, который нам и нужен. Он был запущен вместе с `kube-dns`, это означает, что все в порядке.

Добавление рабочих узлов

Теперь с помощью ранее полученного токена можно добавить в кластер рабочие узлы. Выполните на каждом из узлов (не забудьте `sudo`) следующую команду, подставив токены, которые получили при инициализации Kubernetes на ведущем узле:

```
sudo kubeadm join --token "token" --discovery-token-ca-cert-hash
"discovery token" --ignore-preflight-errors=all
```

На момент написания этой книги (с использованием Kubernetes 1.10) некоторые предварительные проверки завершались неудачно, но это всего лишь ложные срабатывания. На самом деле все в порядке. Можете их пропустить, установив флаг `--ignore-preflight-errors=all`. Надеюсь, когда вы будете читать эти строки, все подобные неувязки уже будут устранены. Вы должны увидеть следующий вывод:

```
[discovery] Trying to connect to API Server "192.168.77.10:6443"
[discovery] Created cluster-info discovery client, requesting info from
"https://192.168.77.10:6443"
[discovery] Requesting info from "https://192.168.77.10:6443" again to
validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate
validates against pinned roots, will use API Server "192.168.77.10:6443"
[discovery] Successfully established connection with API Server
"192.168.77.10:6443"
```

Данный узел присоединился к кластеру:

- * Certificate signing request was sent to master and a response was received.
- * The Kubelet was informed of the new secure connection details.

Чтобы убедиться в этом, выполните на ведущем узле команду `kubectl get nodes`.

В некоторых ситуациях это может не сработать из-за проблем с инициализацией CNI-дополнения.

Создание кластеров в облаке (GCP, AWS и Azure)

Создание локальных кластеров — интересное и важное занятие, относящееся к этапу разработки или локальной отладки. Но как ни крути, платформа Kubernetes предназначена для облачных приложений (запущенных в облаке). Чтобы оставаться масштабируемой, она старается не привязываться к конкретным облачным средам, а взаимодействует с ними через интерфейс `cloud-provider`. Его может реализовать любой облачный провайдер, который хочет поддерживать Kubernetes. Стоит отметить, что на момент выхода версии 1.5 Kubernetes в коде все еще содержатся реализации для разных облачных провайдеров, но в дальнейшем они будут удалены.

Интерфейс `cloud-provider`

Интерфейс `cloud-provider` представляет собой набор типов данных и интерфейсов языка Go. Его определение находится в файле `cloud.go` и доступно по адресу bit.ly/2fq4NbW. Вот как он выглядит:

```
type Interface interface {
    Initialize(clientBuilder controller.ControllerClientBuilder)
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

Здесь все просто. Kubernetes оперирует такими понятиями, как экземпляры, зоны, кластеры и маршруты, требуя указать имя провайдера и открыть доступ к балансировщику нагрузки. Главный интерфейс служит лишь точкой входа. Большинство методов возвращают другие интерфейсы.

Например, интерфейс `Clusters` выглядит крайне простым:

```
type Clusters interface {
    ListClusters() ([]string, error)
    Master(clusterName string) (string, error)
}
```

Метод `ListClusters()` возвращает имена кластеров, метод `Master()` — IP-адрес или DNS-имя ведущего узла.

Другие интерфейсы ненамного сложнее. Весь файл состоит из 214 строк (на момент написания), включая множество комментариев. Это означает, что, если облако поддерживает эти базовые концепции, вам будет несложно реализовать в нем поддержку Kubernetes.

Google Cloud Platform

Google Cloud Platform (GCP) предоставляет стандартную поддержку Kubernetes в виде системы управления контейнерами под названием *Google Kubernetes Engine* (GKE). Для создания и выделения кластеров в GCP не нужно устанавливать Kubernetes — достаточно использовать Google Cloud API. Тот факт, что платформа Kubernetes — это часть GCP, означает, что она всегда будет тесно интегрирована и протестирована и вам не нужно беспокоиться о ее совместимости с интерфейсом cloud-provider.

В целом, если вы планируете основать свою систему на Kubernetes и при этом у вас нет готового кода на других облачных платформах, GCP будет хорошим выбором.

Amazon Web Services

Amazon Web Services (AWS) предоставляет собственный сервис для управления контейнерами под названием ECS, основанный на других технологиях. Но вы все равно можете запускать Kubernetes в AWS, так как это поддерживаемый провайдер (этой теме посвящено множество документации). Выделять ВМ можно самостоятельно с применением *kubeadm*, но я рекомендую обратить внимание на проект *Kops* (*Kubernetes operations*). Он не является частью Kubernetes, но его развитием занимаются те же разработчики. Его можно найти на GitHub (<http://bit.ly/2ft5KA5>).

Kops предоставляет следующие возможности:

- ❑ автоматизированные операции CRUD в кластере Kubernetes для облака (AWS);
- ❑ высокодоступные кластеры Kubernetes;
- ❑ использование модели с синхронизацией состояния для пробных запусков и автоматической идемпотентности;
- ❑ собственную поддержку дополнений утилиты *kubectl*;
- ❑ способность генерировать конфигурацию для Terraform;
- ❑ удобство пользования: вся система основана на простой метамодели, определенной в виде дерева каталогов;
- ❑ простой синтаксис командной строки;
- ❑ поддержку сообщества.

Чтобы создать кластер, нужно внести незначительные изменения в конфигурацию DNS с помощью сервиса `route53`, подготовить объектный контейнер (`bucket`) для хранения настроек кластера и выполнить следующую команду:

```
kops create cluster --cloud=aws --zones=us-east-1c ${NAME}
```

Исчерпывающие инструкции находятся на странице bit.ly/2f7r6EK.

В конце 2017 года сервис AWS присоединился к организации CNCF и объявил о выходе двух больших проектов, связанных с Kubernetes: собственного решения для оркестрации Kubernetes-контейнеров (EKS) и системы, предоставляющей контейнеры по требованию (Fargate).

EKS

EKS (Amazon Elastic Container Service for Kubernetes) — это полностью управляемое высокодоступное решение на основе Kubernetes. Оно содержит три ведущих узла в трех разных зонах доступности и автоматизирует процесс обновления и применения заплаток. Замечательная черта сервиса EKS — то, что он использует обычную версию Kubernetes без каких-либо изменений. Это означает, что вам доступны все стандартные дополнения и инструменты, разработанные сообществом. Также это позволяет применять многокластерный режим, задействуя других облачных провайдеров и/или собственные кластеры Kubernetes, доступные физически.

EKS обеспечивает тесную интеграцию с инфраструктурой AWS. Например, IAM-аутентификация интегрирована в ролевую систему управления доступом Kubernetes (role-based access control, RBAC).

Если вам нужно обращаться к своим ведущим узлам напрямую с собственного частного облака Amazon VPC, можете использовать *PrivateLink*. В этом случае ведущие узлы Kubernetes и конечная точка Amazon EKS будут представлены в Amazon VPC в виде эластичных сетевых интерфейсов с частными IP-адресами.

Еще один важный аспект — специальное дополнение CNI, которое дает возможность компонентам Kubernetes общаться между собой по сети AWS.

Fargate

Fargate позволяет запускать контейнеры напрямую, не беспокоясь о выделении аппаратных ресурсов. Он значительно упрощает процесс управления кластером, но при этом вы теряете определенный контроль. Применяя Fargate, вам достаточно упаковать свое приложение в контейнер, указать требования к процессору и памяти, а также определить сетевые настройки и правила IAM. Fargate может работать поверх ECS и EKS. Это весьма любопытный представитель бессерверных технологий, хотя и не имеет прямого отношения к Kubernetes.

Azure

Платформа *Azure* когда-то предоставляла собственный сервис для управления контейнерами. Вы могли использовать DC/OS на основе Mesos, Docker Swarm

и, конечно же, Kubernetes. Кроме того, самостоятельно можно было выделить кластер Kubernetes (например, с помощью конфигурации желаемого состояния Azure) и затем задействовать утилиту `kubeadm` для его создания. Рекомендуемый подход заключался в применении еще одного стороннего проекта от команды Kubernetes — `kubernetes-anywhere` (<http://bit.ly/2eCS7Ps>), который предоставляет кросс-платформенный способ создания кластеров в облачной среде (как минимум для GCP, AWS и Azure).

Сам процесс довольно прост. Вам нужно установить Docker, утилиты `make` и `kubect1`, имея при этом идентификатор подписчика Azure. Затем клонировать репозиторий `kubernetes-anywhere`, выполнить несколько команд `make` — и кластер готов к использованию.

Исчерпывающее руководство по созданию кластеров в Azure находится по адресу <http://bit.ly/2d56WdA>.

Однако во второй половине 2017 года платформа Azure тоже перешла на сторону Kubernetes и представила AKS-Azure Container Service. Эта система напоминает Amazon EKS, но обладает чуть более глубокой интеграцией.

AKS предоставляет REST API и утилиту командной строки для управления кластерами, но вы можете использовать утилиту `kubect1` или любой другой стандартный инструмент из арсенала Kubernetes.

Далее перечислены некоторые преимущества AKS:

- ☐ автоматический переход на новые версии Kubernetes и применение заплаток;
- ☐ простое масштабирование кластера;
- ☐ панель управления с автоматическим устранением неисправностей, размещенная на ведущих узлах;
- ☐ экономия ресурсов — клиент платит только за запущенные узлы.

В данном разделе мы рассмотрели интерфейс `cloud-provider` и познакомились с рекомендациями по созданию кластеров Kubernetes на разных облачных платформах. Эта индустрия все еще находится на раннем этапе развития, ее инструментарий стремительно совершенствуется. Я верю, что в какой-то момент все проекты вроде `kubeadm`, `kops`, `Kargo` и `kubernetes-anywhere` придут к общему, унифицированному, простому решению по созданию кластеров Kubernetes.

Alibaba Cloud

На рынке облачных услуг появился новый перспективный конкурент — китайский сервис *Alibaba Cloud*. Он во многом похож на AWS, хотя его англоязычная документация оставляет желать лучшего. Я уже разворачивал промышленное приложение в Ali Cloud, но оно не использовало кластеры Kubernetes. И хотя поддержка Kubernetes заявлена официально, документация доступна лишь на китайском языке. Мне удалось найти одну англоязычную публикацию на форуме Alibaba, в которой описывается процесс разворачивания кластера Kubernetes в Ali Cloud: www.alibabacloud.com/forum/read-830.

Создание аппаратного кластера с нуля

В предыдущем разделе мы обсудили запуск Kubernetes в облаке. Это наиболее распространенный сценарий применения данной платформы, но в определенных ситуациях для ее работы имеет смысл использовать «голое железо». Я не стану останавливаться на том, где лучше размещать свои серверы — у себя или у внешнего провайдера, так как это отдельная история. Если вы уже обслуживаете множество локальных серверов, вам должно быть виднее.

Сценарии использования «голого железа»

Аппаратный кластер — вещь непростая, особенно если вы обслуживаете его самостоятельно. Некоторые компании, такие как Platform 9, предоставляют коммерческую поддержку аппаратных кластеров Kubernetes, но эти решения пока что сложно назвать зрелыми. Существует также развитая система с открытым кодом, Kubespray, которая позволяет развертывать кластеры Kubernetes промышленного уровня на «голом железе», в AWS, GCE, Azure и OpenStack.

Вот несколько ситуаций, в которых это может иметь смысл.

- ❑ **Ограниченный бюджет.** Если в вашем распоряжении уже имеется реальная крупномасштабная инфраструктура, использовать ее для работы Kubernetes может оказаться намного дешевле.
- ❑ **Небольшие сетевые задержки.** Если ваши узлы должны взаимодействовать с минимальной задержкой, накладные расходы на виртуальные машины могут оказаться чрезмерными.
- ❑ **Правовые требования.** Некоторые нормы законодательства исключают использование облачных провайдеров.
- ❑ **Полный контроль над аппаратным обеспечением.** Несмотря на богатый выбор серверов, облачные провайдеры не удовлетворяют все возможные потребности.

Когда создание аппаратного кластера имеет смысл

Создание кластера с нуля связано с существенными сложностями. Kubernetes — непростая система. В Интернете имеется множество руководств по настройке аппаратных кластеров, но многие из них быстро устаревают, так как экосистема движется вперед семимильными шагами.

Этот путь следует рассматривать в том случае, если у вас есть ресурсы и время для отладки всего стека технологий на каждом его уровне. Большинство неполадок обычно связано с сетью, но файловые системы и драйверы для хранилищ данных

тоже могут доставить немало проблем, равно как и несовместимость компонентов и их версий, таких как Docker (или gkt, если у вас хватит смелости), образы контейнеров, ОС, ее ядро, различные дополнения и инструменты, которые вы используете, и сама платформа Kubernetes.

Процесс создания кластера

Впереди у вас много работы. Вот список дел, которыми вам предстоит заняться.

- ☐ Реализация интерфейса для собственного облачного провайдера или использование обходных путей.
- ☐ Выбор сетевой модели и способа ее реализации (непосредственное встраивание или применение дополнения CNI).
- ☐ Использование сетевых правил или отказ от них.
- ☐ Выбор образов для системных компонентов.
- ☐ Модели безопасности и SSL-сертификаты.
- ☐ Администраторские полномочия.
- ☐ Шаблоны таких компонентов, как API-сервер, контроллер репликации и планировщик.
- ☐ Сервисы кластера, такие как DNS, журналирование, мониторинг и веб-интерфейс.

Чтобы получить четкое представление о том, что именно требуется для создания кластера с нуля, я рекомендую почитать руководство на сайте Kubernetes (<http://bit.ly/1ToR9EC>).

Использование инфраструктуры частного виртуального облака

Если в вашем случае имеет смысл применять аппаратные кластеры, но нет квалифицированного персонала или желания заниматься обслуживанием аппаратной инфраструктуры, обратите внимание на частное облако вроде OpenStack (например, в сочетании с stackube). Если вы чувствуете себя комфортней на более высоком уровне абстракции, вам может подойти облачная платформа от компании Mirantis, построенная на основе OpenStack и Kubernetes.

В этом разделе рассмотрен вариант с использованием кластеров Kubernetes на «голом железе». Мы обсудили разные сценарии, которые требуют такого подхода, и отметили вызовы и сложности, которые с ним связаны.

Bootkube. Вас также может заинтересовать *Bootkube*. Эта система способна запускать кластеры Kubernetes на вашей собственной инфраструктуре. Это означает, что большинство компонентов работают в виде подов и доступны для управления, мониторинга и обновления с использованием тех же инструментов и методик,

которые вы применяете в контейнеризованных приложениях. Этот подход имеет существенные преимущества, позволяя упростить разработку и обслуживание кластеров Kubernetes.

Резюме

В этой главе состоялось знакомство с практической стороной создания кластеров. Мы создали одноузловой и многоузловой кластеры с помощью Minikube и соответственно kubectl. Затем рассмотрели различные варианты работы с Kubernetes в облачных платформах. В конце затронули тему создания кластеров Kubernetes на «голом железе» и связанные с этим сложности. Текущая ситуация крайне изменчива: основные компоненты постоянно развиваются, инструментарий все еще незрелый, и у каждой среды есть свои особенности. Подготовка кластера Kubernetes — нетривиальная задача, но, приложив определенные усилия и уделив внимание деталям, вы вскоре сможете ее освоить.

В следующей главе мы обсудим важные темы — мониторинг, журналирование и отладку. Запустив свой кластер и начав разворачивать на нем рабочие задания, вы должны убедиться в том, что он работает как следует и соответствует заданным критериям. Это требует постоянного внимания и умения реагировать на различные неполадки, которые встречаются в реальном мире.

3

Мониторинг, журналирование и решение проблем

В предыдущей главе вы научились разворачивать Kubernetes в разных средах, поэкспериментировали с различными инструментами и создали парочку кластеров. Но все это лишь полдела. Запустив кластер, вы должны убедиться в том, что он находится в рабочем состоянии, все его компоненты на месте и как следует сконфигурированы и вы развернули достаточно ресурсов для удовлетворения поставленных требований. Реакция на сбои, отладка и решение проблем — все это важная часть управления любой сложной системой, и Kubernetes не исключение.

В этой главе рассмотрим следующие темы.

- ❑ Мониторинг с помощью Heapster.
- ❑ Анализ производительности с помощью панели управления Kubernetes.
- ❑ Централизованное ведение журнала.
- ❑ Обнаружение проблем на уровне узла.
- ❑ Примеры решения проблем.
- ❑ Использование Prometheus.

Прочитав эту главу, вы станете четко понимать различные способы мониторинга кластеров Kubernetes и узнаете, как обращаться к журнальным файлам и анализировать их. Вам будет достаточно лишь взглянуть на кластер, чтобы убедиться в его исправности. Вместе с тем вы сможете проводить методическую диагностику, выявлять и устранять проблемы.

Мониторинг Kubernetes с помощью Heapster

Heapster — проект от команды Kubernetes, который предоставляет полноценное решение для мониторинга кластеров. Оно запускается в виде пода (естественно), поэтому им можно управлять непосредственно через Kubernetes. Heapster поддерживает кластеры на основе Kubernetes и CoreOS и отличается модульной и гибкой архитектурой. Эта система собирает операционные показатели и события с каждого узла в кластере и сохраняет их в постоянной базе данных с четко определенной

структурой и программируемым доступом, после чего их можно визуализировать. Heapster поддерживает разные хранилища (или стоки в терминологии Heapster) и клиентские компоненты для визуализации. Самая популярная комбинация состоит из InfluxDB на сервере и Grafana на клиенте. Google Cloud Platform обеспечивает интеграцию Heapster со своей службой мониторинга. Существуют также другие, менее известные серверные модули:

- ❑ Log;
- ❑ Google Cloud monitoring;
- ❑ Google Cloud logging;
- ❑ Hawkular-Metrics (только показатели);
- ❑ OpenTSDB;
- ❑ Monasca (только показатели);
- ❑ Kafka (только показатели);
- ❑ Riemann (только показатели);
- ❑ Elasticsearch.

Вы можете использовать сразу несколько серверных модулей, указав подходящие стоки в командной строке:

```
--sink=log --sink=influxdb:http://monitoring-influxdb:80/
```

cAdvisor. Система cAdvisor входит в состав утилиты kubelet, которая работает на всех узлах. Она собирает информацию о нагрузке на процессор/ядра, память, сеть и файловую систему каждого контейнера. cAdvisor предоставляет простой пользовательский интерфейс на порте 4194. Но наиболее важно с точки зрения Heapster то, что все собранные ею данные доступны через kubelet. Heapster собирает их со всех узлов и сохраняет на своем сервере для дальнейших анализа и визуализации.

Пользовательский интерфейс cAdvisor позволяет быстро проверить корректность настройки того или иного узла, например, во время создания нового кластера, пока вы еще не подключили Heapster (рис. 3.1).

Установка Heapster

Компоненты Heapster уже могут быть установлены в вашем кластере Kubernetes. Если это не так, можете установить их несколькими простыми командами. Сначала клонируем репозиторий Heapster:

```
> git clone https://github.com/kubernetes/heapster.git  
> cd heapster
```

В ранних версиях Kubernetes система Heapster по умолчанию предоставляла свои сервисы в виде компонентов NodePort. Теперь же для этого используются

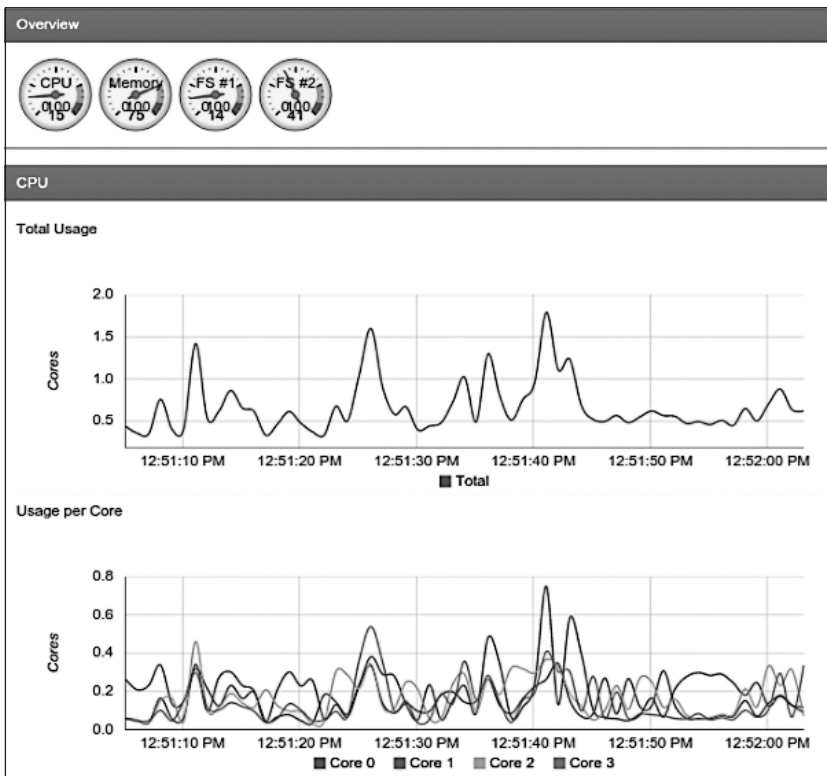


Рис. 3.1

компоненты ClusterIP, доступные только внутри кластера. Чтобы к ним можно было обращаться локально, я добавил в описание каждого сервиса внутри `deploy/kube-config/influxdb` тип `NodePort`. Вот так, например, выглядит файл `deploy/kubeconfig/influxdb/influxdb.yaml`:

```
> git diff deploy/kube-config/influxdb/influxdb.yaml
diff -git a/deploy/kube-config/influxdb/influxdb.yaml b/deploy/kubeconfig/
influxdb/influxdb.yaml
index 29408b81..70f52d2c 100644
--- a/deploy/kube-config/influxdb/influxdb.yaml
+++ b/deploy/kube-config/influxdb/influxdb.yaml
@@ -33,6 +33,7 @@ metadata:
   name: monitoring-influxdb
   namespace: kube-system
 spec:
+ type: NodePort
   ports:
   - port: 8086
     targetPort: 8086
```

Аналогичное изменение сделано в файле `deploy/kube-config/influxdb/grafana.yaml`: в нем строка `+ type: NodePort` была закомментирована, поэтому я просто убрал символ комментария. Теперь можем установить InfluxDB и Grafana:

```
> kubectl create -f deploy/kube-config/influxdb
```

Вы должны увидеть следующий вывод:

```
deployment "monitoring-grafana" created
service "monitoring-grafana" created
serviceaccount "heapster" created
deployment "heapster" created
service "heapster" created
deployment "monitoring-influxdb" created
service "monitoring-influxdb" created
```

Хранилище InfluxDB

InfluxDB — современная, полноценная, распределенная база данных с поддержкой временных рядов. Она отлично подходит и широко применяется для централизованного хранения показателей и журнальных записей. Это рекомендуемое хранилище для Heapster (вне Google Cloud Platform). Единственный недостаток InfluxDB связан с кластеризацией — высокая доступность поддерживается только в платной версии для предприятий.

Структура хранилища

Структура хранилища InfluxDB определяет информацию, которую сохраняет Heapster, с возможностью последующего поиска и графического отображения. Показатели разделены на несколько категорий, которые называются измерениями. Вы можете запрашивать как целые категории в виде объединенных измерений, так и отдельные показатели, доступные в виде полей. Именование выполняется по принципу `<категория>/<имя_показателя>` (кроме времени работы, у которого лишь один показатель). Если вы уже имели дело с SQL, относитесь к измерениям как к таблицам. Каждый показатель относится к определенному контейнеру и содержит следующую метainформацию:

- ❑ `pod_id` — уникальный идентификатор пода;
- ❑ `pod_name` — имя пода, заданное пользователем;
- ❑ `pod_namespace` — пространство имен пода;
- ❑ `container_base_image` — базовый образ для контейнера;
- ❑ `container_name` — имя контейнера, заданное пользователем, или полное имя `cgroup` в случае с системными контейнерами;
- ❑ `host_id` — идентификатор узла, заданный пользователем или относящийся к определенному облачному провайдеру;
- ❑ `hostname` — имя сетевого узла, на котором работает контейнер;

- ❑ `labels` — список пользовательских меток, разделенных запятыми, в формате `ключ:значение`;
- ❑ `namespace_id` — уникальный идентификатор пространства имен пода;
- ❑ `resource_id` — уникальный идентификатор, позволяющий различать метки одного типа — например, разделы файловой системы в измерении `filesystem/usage`.

Далее представлены все показатели, сгруппированные по категориям. Как видите, их список довольно обширен.

Центральный процессор

Показатели центрального процессора:

- ❑ `cpu/limit` — жесткий лимит в миллиардах;
- ❑ `cpu/node_capacity` — вычислительная мощность узла;
- ❑ `cpu/node_allocatable` — доступность ЦПУ для узла;
- ❑ `cpu/node_reservation` — часть ЦПУ, зарезервированная для узла;
- ❑ `cpu/node_utilization` — загруженность части ЦПУ, выделенной для узла;
- ❑ `cpu/request` — запрос ЦПУ (гарантированный объем ресурсов) в миллиардах;
- ❑ `cpu/usage` — совокупная загруженность всех ядер ЦПУ;
- ❑ `cpu/usage_rate` — совокупная загруженность всех ядер ЦПУ в миллиардах.

Файловая система

Показатели файловой системы:

- ❑ `filesystem/usage` — общее количество байтов, использованное в файловой системе;
- ❑ `filesystem/limit` — общий размер файловой системы в байтах;
- ❑ `filesystem/available` — количество байтов, все еще доступных в файловой системе.

Память

Показатели памяти:

- ❑ `memory/limit` — жесткий лимит на память в байтах;
- ❑ `memory/major_page_faults` — количество аппаратных отказов страницы;
- ❑ `memory/major_page_faults_rate` — количество аппаратных отказов страницы в секунду;
- ❑ `memory/node_capacity` — память, занятая узлом;
- ❑ `memory/node_allocatable` — память, выделенная узлу;
- ❑ `memory/node_reservation` — часть памяти, зарезервированная для узла;

- ❑ `memory/node_utilization` — использование части памяти, выделенной узлу;
- ❑ `memory/page_faults` — количество отказов страницы;
- ❑ `memory/page_faults_rate` — количество отказов страницы в секунду;
- ❑ `memory/request` — запрос памяти (гарантированный объем ресурсов) в байтах;
- ❑ `memory/usage` — общее использование памяти;
- ❑ `memory/working_set` — общее использование рабочего набора, то есть задействованной памяти, которую ядро не может легко освободить.

Сеть

Показатели сети:

- ❑ `network/rx` — совокупное количество байтов, полученных по сети;
- ❑ `network/rx_errors` — совокупное количество ошибок при получении данных по сети;
- ❑ `network/rx_errors_rate` — количество ошибок в секунду при получении данных по сети;
- ❑ `network/rx_rate` — скорость получения данных по сети в байтах в секунду;
- ❑ `network/tx` — совокупное количество байтов, отправленных по сети;
- ❑ `network/tx_errors` — совокупное количество ошибок при отправке данных по сети;
- ❑ `network/tx_errors_rate` — количество ошибок в секунду при отправке данных по сети;
- ❑ `network/tx_rate` — скорость отправки данных по сети в байтах в секунду.

Время работы

Время работы (`uptime`) — это количество миллисекунд, прошедших с момента запуска контейнера.

Работайте с InfluxDB напрямую, если знакомы с этой БД. Можно подключиться к ней по API или воспользоваться веб-интерфейсом. Чтобы найти ее порт и конечную точку, введите следующую команду:

```
> k describe service monitoring-influxdb -namespace=kube-system | grep
NodePort
Type:                NodePort
NodePort:            <unset> 32699/TCP
```

Теперь можете познакомиться с веб-интерфейсом InfluxDB, используя HTTP-порт. Вы должны указать порт, который будет ссылаться на API. Поля `Username` (Имя пользователя) и `Password` (Пароль) по умолчанию содержат значение `root` (рис. 3.2).

Закончив с настройками, выберите базу данных (в правом верхнем углу). База данных Kubernetes называется `k8s`. Теперь из InfluxDB можно получить показатели, используя язык запросов этой БД.

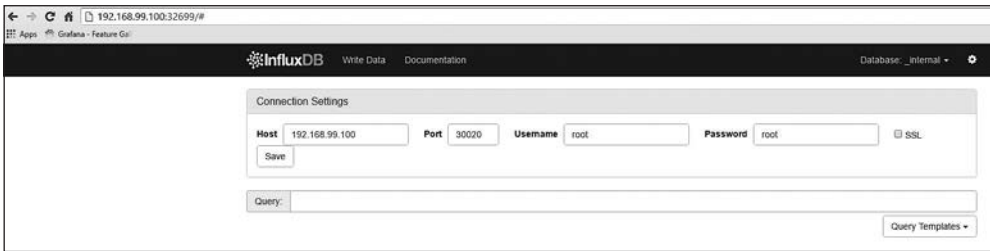


Рис. 3.2

Визуализация с помощью Grafana

Сервис Grafana работает в отдельном контейнере и предоставляет многофункциональную панель управления, которая может использовать InfluxDB в качестве источника данных. Чтобы найти нужный порт, введите следующую команду:

```
k describe service monitoring-influxdb -namespace=kube-system | grep
NodePort
Type:                NodePort
NodePort:            <unset> 30763/TCP
```

Теперь можете подключаться к веб-интерфейсу Grafana на этом порте. Первым делом нужно сделать так, чтобы источник данных ссылался на сервер InfluxDB (рис. 3.3).

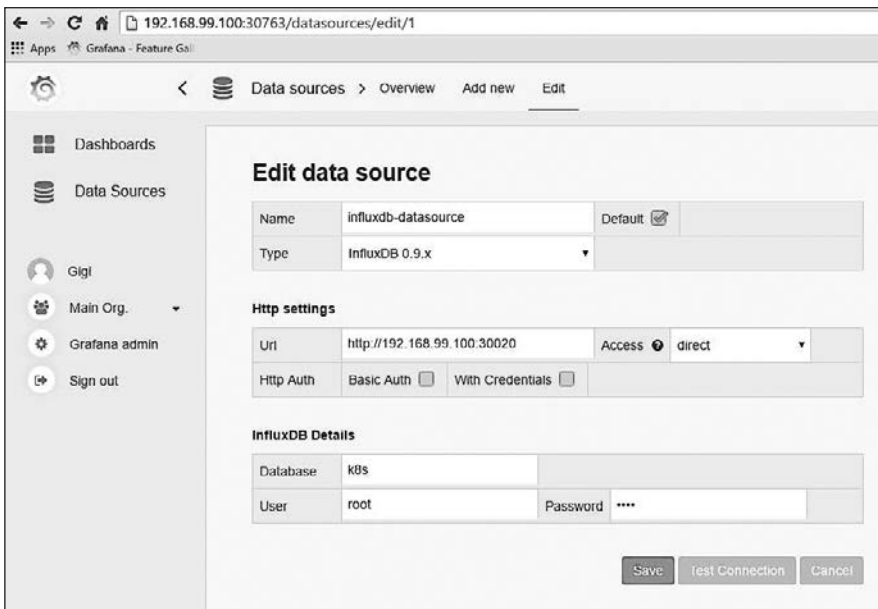


Рис. 3.3

Не забудьте проверить соединение и ознакомиться с различными возможностями панели управления. Вам будут доступны несколько стандартных панелей, но Grafana позволяет подстраивать их под свои нужды.

Анализ производительности с помощью панели управления

Панель управления Kubernetes — это мой любимый инструмент на случай, когда просто хочется узнать, что происходит в кластере. Тому есть несколько причин.

- ❑ Она встроена в платформу (всегда синхронизирована и проверена на совместимость с Kubernetes).
- ❑ Она быстрая.
- ❑ Она имеет интуитивно понятный иерархический интерфейс, охватывающий все уровни — от кластера до отдельных контейнеров.
- ❑ Ее не нужно дополнительно настраивать.

Инструменты Heapster, InfluxDB и Grafana лучше подходят для сложных настраиваемых представлений и запросов, в то время как панель управления Kubernetes имеет набор готовых страниц, которые обычно удовлетворяют 80–90 % любых потребностей.

С помощью панели управления можно развертывать приложения и создавать любые ресурсы Kubernetes — достаточно лишь загрузить подходящий YAML- или JSON-файл. Но поскольку такой подход к управлению инфраструктурой считается нежелательным, не стану его здесь описывать. Это может быть удобно во время экспериментов с тестовым кластером, но для изменения состояния системы я предпочитаю использовать командную строку. А вы применяйте то, что вам удобно.

Сначала найдем порт:

```
k describe service kubernetes-dashboard --namespace=kube-system | grep
NodePort
Type:                NodePort
NodePort:            <unset> 30000/TCP
```

Представление верхнего уровня

Панель управления разделена на две части: слева находится иерархия (ее можно спрятать, щелкнув на значке в виде гамбургера), а справа — динамическое содержимое, зависящее от контекста. Пройдитесь по иерархическому меню, чтобы получить нужную информацию.

Вам доступны несколько категорий верхнего уровня:

- ❑ Cluster (Кластер);
- ❑ Overview (Обзор);

- ☐ Workloads (Рабочие задания);
- ☐ Discovery and load balancing (Обнаружение и балансировка нагрузки);
- ☐ Config and storage (Конфигурация и хранилище).

Можно фильтровать информацию по определенному пространству имен или выбрать сразу все категории.

Кластер

Представление Cluster (Кластер) состоит из пяти разделов: Namespaces (Пространства имен), Nodes (Узлы), Persistent Volumes (Постоянные тома), Roles (Роли) и Storage Classes (Классы хранилищ) (рис. 3.4).

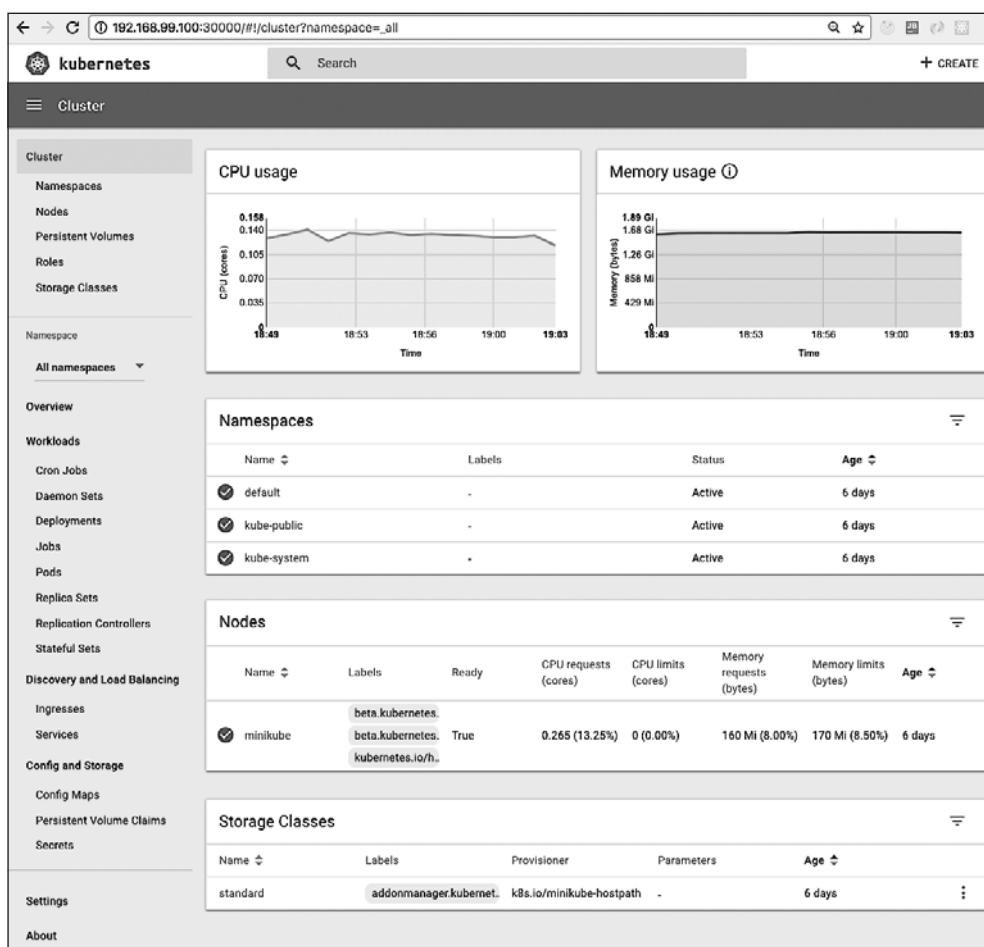


Рис. 3.4

Вы сразу же получите множество информации: степень использования процессора и памяти на всех узлах, доступные пространства имен, их состояние (Status) и возраст (Age). Можете узнать возраст (Age) и метки (Labels) каждого узла, увидеть постоянные тома и роли (если таковые имеются), а также классы хранилищ (в данном случае это просто путь к узлу).

Если углубиться еще и щелкнуть на узле `minikube`, то получите аккуратную круговую диаграмму с подробными сведениями об этом узле и занятых ресурсах. Это незаменимая функция при решении проблем с производительностью. Если узлу не хватает ресурсов, он не сможет удовлетворить потребности своих подов (рис. 3.5).

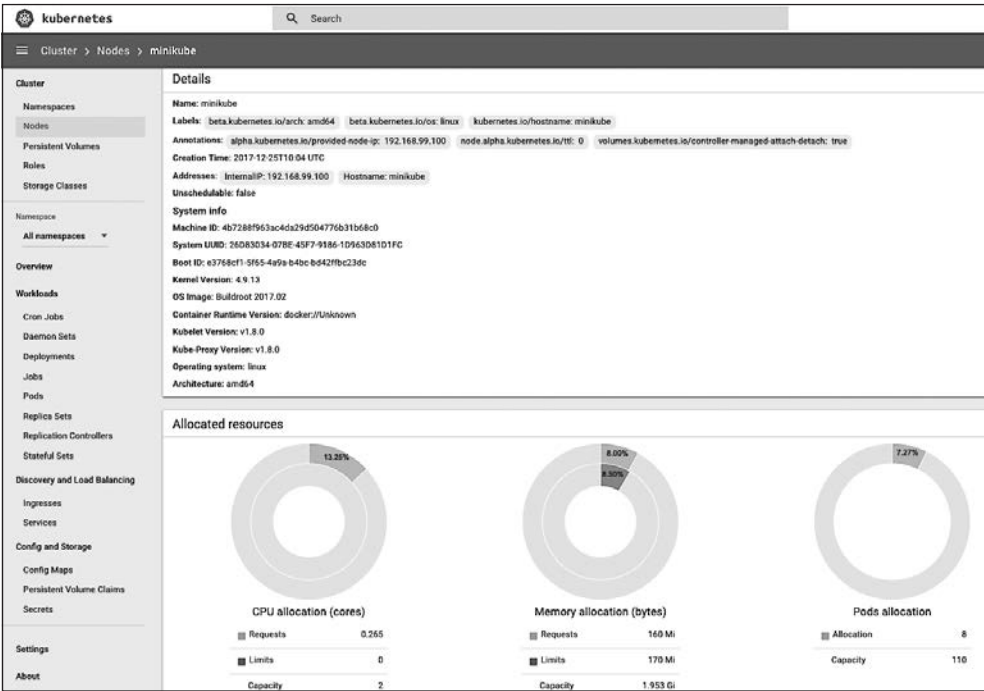


Рис. 3.5

Опустившись вниз, вы увидите еще более интересную информацию. Обратите внимание на панель **Conditions** (Условия). Это прекрасная краткая сводка о нагрузке на память и диск на отдельно взятом узле (рис. 3.6).

Там же находятся панели **Pods** (Поды) и **Events** (События). Мы обсудим поды в следующем разделе.

Рабочие задания

Это основная категория. Она охватывает множество типов ресурсов Kubernetes, таких как **CronJobs** (Планируемые задачи), **Daemon Sets** (Наборы демонов), **Deployments** (Развернутые приложения), **Jobs** (Задания), **Pods** (Поды), **Replica Sets** (Наборы реплик),

Conditions					
Type	Status	Last heartbeat time	Last transition time	Reason	Message
OutOfDisk	False	7 seconds	6 days	KubeletHasSufficientDisk	kubelet has sufficient disk space available
MemoryPressure	False	7 seconds	6 days	KubeletHasSufficientMemory	kubelet has sufficient memory available
DiskPressure	False	7 seconds	6 days	KubeletHasNoDiskPressure	kubelet has no disk pressure
Ready	True	7 seconds	4 hours	KubeletReady	kubelet is posting ready status

Pods							
Name	Namespace	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
heapster-d7688d788-v1hfx	kube-system	minikube	Running	0	34 minutes	0	21.012 Mi
monitoring-influxdb-77bd46594b-zv	kube-system	minikube	Running	0	34 minutes	0	22.719 Mi
monitoring-grafana-5d967d96d-gh	kube-system	minikube	Running	0	34 minutes	0	11.426 Mi
echo-557f84bf4f-p5fvq	default	minikube	Running	1	5 days	0	2.555 Mi
kube-dns-86f6f55d65-zfvnz	kube-system	minikube	Running	6	6 days	0.001	34.215 Mi
kubernetes-dashboard-q5xcm	kube-system	minikube	Running	2	6 days	0	31.180 Mi
storage-provisioner	kube-system	minikube	Running	2	6 days	0	14.281 Mi
kube-addon-manager-minikube	kube-system	minikube	Running	2	6 days	0.021	33.398 Mi

Рис. 3.6

Replication Controllers (Контроллеры репликации) и Stateful Sets (Наборы с состоянием). Вы можете подробно изучить любую из этих категорий. На рис. 3.7 показан верхний уровень представления Workloads (Рабочие задания) для стандартного пространства имен, в котором пока что развернут лишь сервис echo. Здесь вы увидите панели Deployments (Развернутые приложения), Replica Sets (Наборы реплик) и Pods (Поды).

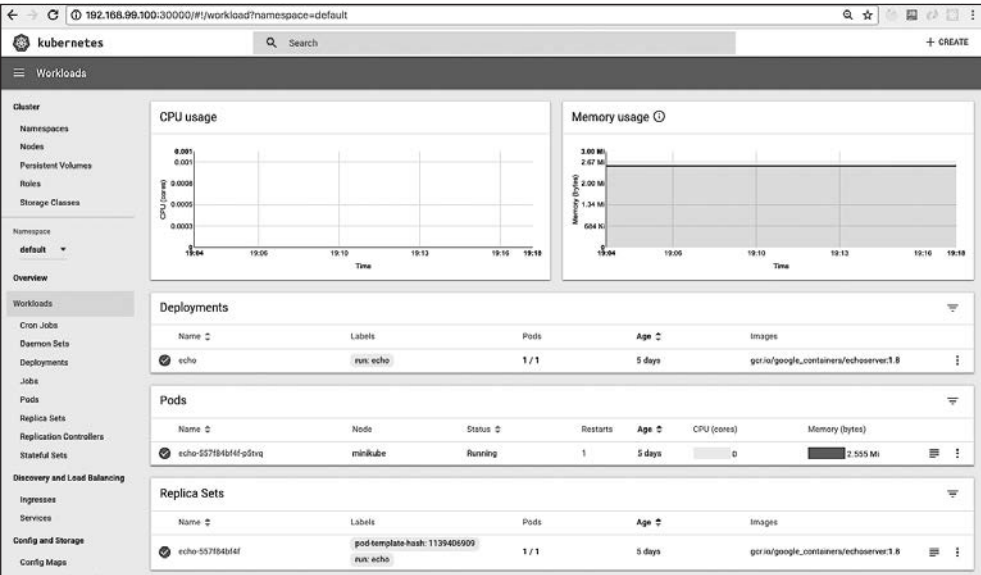


Рис. 3.7

Подключим все пространства имен и откроем подкатегорию Pods (Поды). Это очень полезное представление. В каждой строке показано, запущен под или нет, сколько раз он был перезапущен, указан его IP-адрес. Внизу находятся небольшие аккуратные графики использования процессора и памяти (рис. 3.8).

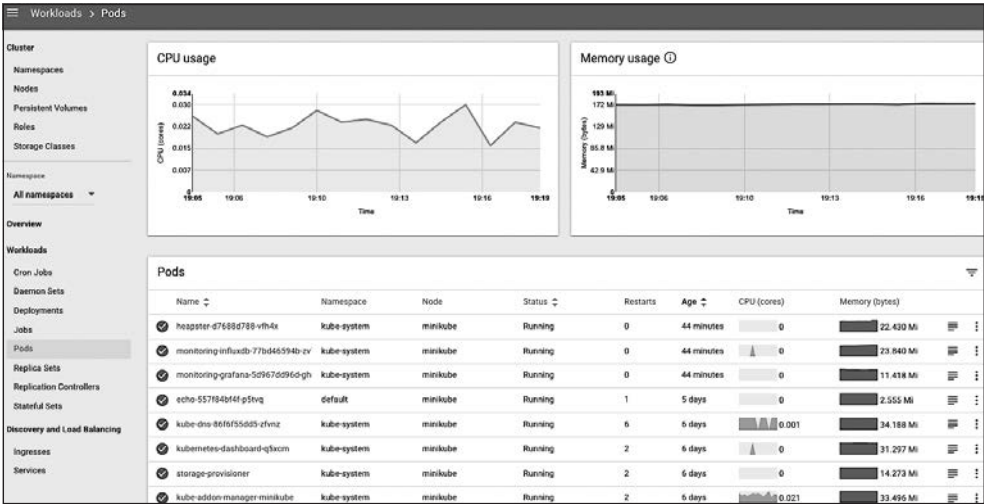


Рис. 3.8

Вы также можете просмотреть журналы любого пода, щелкнув на значке с текстом (второй справа). Проверим журнал оболочки InfluxDB. Похоже, все в порядке и Heapster успешно в него записывает (рис. 3.9).

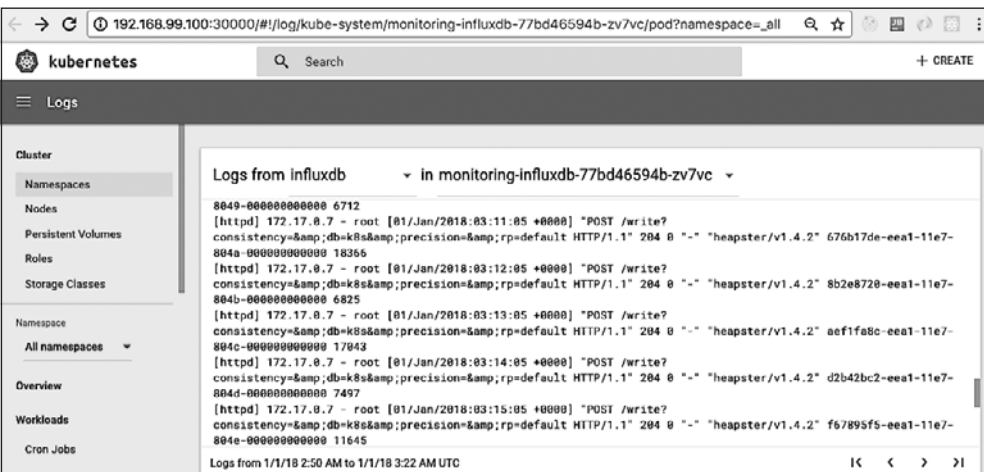


Рис. 3.9

Есть еще одна степень детализации, до которой мы еще не дошли, — уровень контейнера. Щелкнем на поде `kubedns`. Вы увидите следующую страницу, на которой перечислены отдельные контейнеры и их команды запуска. Можно также просмотреть их журналы (рис. 3.10).

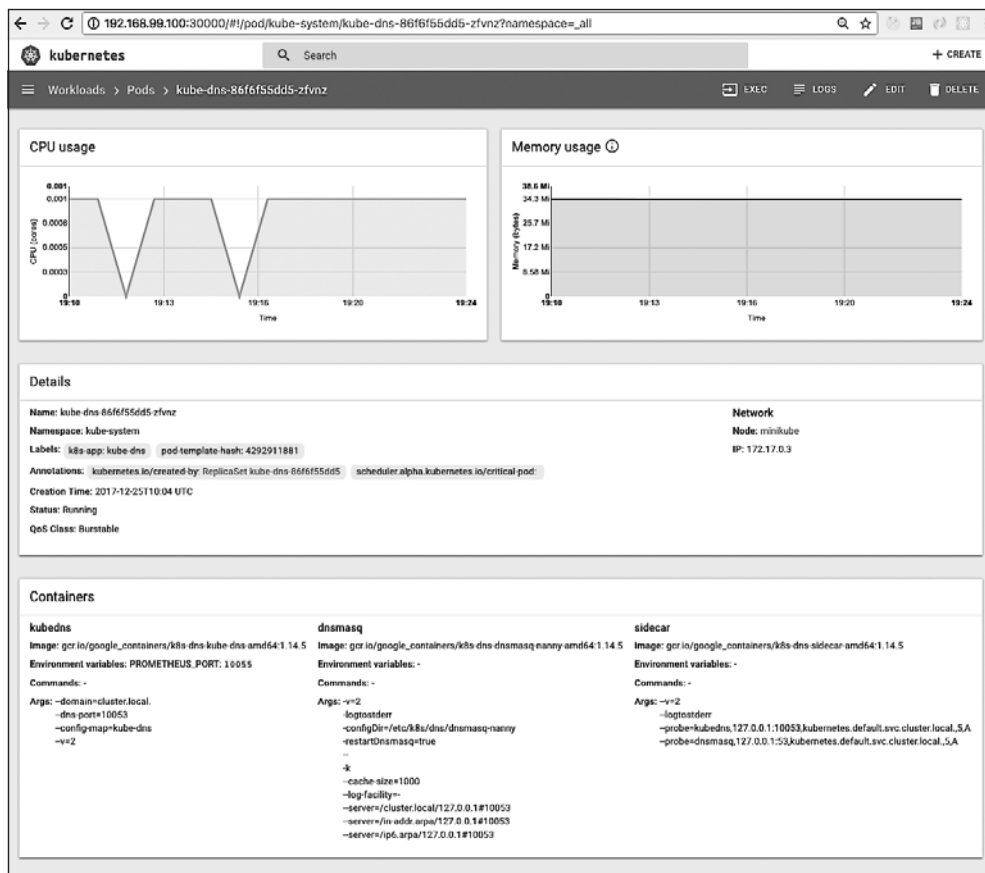


Рис. 3.10

Обнаружение и балансировка нагрузки

Эта категория часто служит отправной точкой. Панель **Services** (Сервисы) — это публичный интерфейс вашего кластера Kubernetes. Если сервисы испытывают серьезные сложности, это неизбежно отразится на ваших пользователях (рис. 3.11).

Вы можете углубиться еще больше и щелкнуть на сервисе, чтобы получить некоторую информацию о нем (самое важное — селектор метки) и список его подов.

Name	Namespace	Labels	Cluster IP	Internal endpoints	External endpoints	Age
monitoring-grafana	kube-system	kubernetes.io/cluster: kubernetes.io/name:	10.99.38.116	monitoring-grafana.ku monitoring-grafana.ku	-	48 minutes
monitoring-influxdb	kube-system	kubernetes.io/cluster: kubernetes.io/name: task: monitoring	10.97.11.229	monitoring-influxdb.ku monitoring-influxdb.ku	-	48 minutes
heapster	kube-system	kubernetes.io/cluster: kubernetes.io/name: task: monitoring	10.105.93.217	heapster.kube-system: heapster.kube-system:	-	48 minutes
kubelet	kube-system	k8s-app: kubelet	None	kubelet.kube-system:1 kubelet.kube-system:0	-	4 hours
echo	default	run: echo	10.100.78.178	echo:8080 TCP echo:30127 TCP	-	5 days
kube-dns	kube-system	addonmanager.kube. k8s-app: kube-dns kubernetes.io/name:	10.96.0.10	kube-dns.kube-system kube-dns.kube-system kube-dns.kube-system	-	6 days
kubemetes-dashboard	kube-system	addonmanager.kube. app: kubemetes-das- kubernetes.io/minik- kubernetes.io/minik-	10.101.107.20	kubemetes-dashboard kubemetes-dashboard	-	6 days
kubemetes	default	component: apiserve provider: kubemetes	10.96.0.1	kubemetes:443 TCP kubemetes:0 TCP	-	6 days

Рис. 3.11

Добавление централизованного журналирования

Централизованное журналирование на уровне кластера — это фундаментальный элемент любой системы со значительным количеством узлов, подов или контейнеров. Прежде всего, было бы непрактично просматривать журналы каждого пода или каждого контейнера по отдельности. Вы не смогли бы получить общее представление о системе, и вам пришлось бы перебирать слишком много сообщений. Требуется решение, которое позволяло бы агрегировать и легко фильтровать журнальные записи. Еще одна причина состоит в том, что контейнеры являются временными сущностями. Наборы реплик и контроллеры репликации часто просто удаляют неисправные поды и запускают вместо них новые, теряя при этом важные журнальные данные. Делегируя журналирование центральному сервису, вы можете сохранить информацию, которая способна оказаться незаменимой при решении проблем.

Планирование централизованного журналирования

В принципе, в централизованном ведении журнала нет ничего сложного. На каждом узле запускается специальный агент, который перехватывает все журнальные записи из всех подов и контейнеров на этом узле и передает их вместе с определенными метаданными в центральный репозиторий для дальнейшего безопасного хранения.

Как упоминалось ранее, платформа Google предоставляет систему GKE, в которую уже интегрирован сервис централизованного журналирования. На других платформах часто используют сочетание решений fluentd, Elasticsearch и Kibana. Для них выпущено официальное дополнение fluentd-elasticsearch, которое находится по адресу <http://bit.ly/2f6MF5b>. Оно устанавливается в виде сервисов для Elasticsearch и Kibana, при этом на каждом узле разворачивается агент fluentd.

Fluentd

Fluentd — это унифицированный слой журналирования, который размещается между источниками и произвольными стоками данных и обеспечивает передачу журнальных записей из пункта А в пункт Б. У Kubernetes есть дополнение с образом Docker, которое разворачивает агент fluentd, способный считывать журнальные файлы различных программ, связанных с Kubernetes, например Docker, etcd и Kube. Оно также внедряет метки в каждую журнальную запись, чтобы позже их можно было фильтровать. Далее показан фрагмент файла `fluentd-es-configmap.yaml`:

```
# Example:
# 2016/02/04 06:52:38 filePurge: successfully removed file
/var/etcd/data/member/wal/00000000000006d0-00000000010a23d1.wal
<source>
  type tail
  # Not parsing this, because it doesn't have anything particularly
  useful to
  # parse out of it (like severities).
  Format none
  path /var/log/etcd.log
  pos_file /var/log/es-etcd.log.pos
  tag etcd
</source>
```

Elasticsearch

Elasticsearch — это отличное документное хранилище с возможностью полнотекстового поиска. Благодаря высокой скорости, надежности и масштабируемости оно пользуется большой популярностью в приложениях уровня предприятия. Elasticsearch входит в состав дополнения Kubernetes для организации централизованного журналирования и разворачивается в виде сервиса из образа Docker. Стоит отметить, что полноценный промышленный кластер для этого хранилища, который разворачивается внутри кластера Kubernetes, требует наличия собственных узлов (ведущих, клиентских и для хранения данных). В крупномасштабных и высокодоступных кластерах Kubernetes централизованное ведение журнала кластеризуется само по себе. Elasticsearch также поддерживает автоматическое обнаружение. Решение уровня предприятия доступно по адресу github.com/pires/kubernetes-elasticsearch-cluster.

Kibana

Kibana — это своеобразная надстройка над Elasticsearch, которая используется для визуализации данных, хранящихся и проиндексированных в этой базе данных, и взаимодействия с ними. Она тоже входит в состав дополнения и устанавливается в виде сервиса. Шаблон ее Docker-файла доступен по адресу <http://bit.ly/2lwmtpc>.

Обнаружение неисправностей на узлах

В концепции Kubernetes единицей работы является под. Однако выполнение подов планируется на узлах. Когда речь идет о мониторинге и надежности, именно узлы требуют наиболее пристального внимания, так как экземплярами подов занимается сама платформа Kubernetes (с помощью планировщика и контроллеров репликации). Узлы могут испытывать различные проблемы, о которых Kubernetes ничего не известно. В итоге поды будут и дальше развертываться на неисправных узлах, что способно помешать их корректной работе.

Далее перечислены проблемы, которые может испытывать внешне нормально функционирующий узел:

- ☐ неисправный процессор;
- ☐ неисправная память;
- ☐ неисправный жесткий диск;
- ☐ взаимное блокирование в ядре;
- ☐ поврежденная файловая система;
- ☐ проблемы с демоном Docker.

Kubelet и cAdvisor не отслеживают подобные неполадки, поэтому нам нужно другое решение. Для начала познакомимся с Node Problem Detector.

Node Problem Detector

Node Problem Detector — это под, запущенный на каждом узле. Перед ним стоят непростые задачи. Он должен обнаруживать различные неисправности в разных средах, на различных аппаратных платформах и в операционной системе. Ему нужно быть достаточно надежным, чтобы самому не пострадать от таких проблем, иначе он не сможет о них сообщить, и при этом иметь относительно низкие накладные расходы, чтобы не нагружать ведущий узел. Кроме того, он должен работать на всех узлах. Специально для этого в Kubernetes добавлена новая подсистема под названием DaemonSet. Исходный код находится по адресу <https://github.com/kubernetes/node-problem-detector>.

DaemonSet

DaemonSet — это под для всех узлов. Если ваш кластер содержит определение DaemonSet, на каждый новый узел автоматически будет добавляться соответствующий экземпляр пода. Это контроллер репликации, который разворачивает по одному поду на каждом узле. Сервис Node Problem Detector определен как DaemonSet, что идеально соответствует его требованиям. Для более тонкого контроля за планированием DaemonSet можно использовать такие параметры, как принадлежность, непринадлежность и ограничения.

Демоны для обнаружения проблем

Сервис Node Problem Detector имеет одну проблему (простите за каламбур): список неисправностей, которые он должен находить, слишком обширный. Попытка вставить все это в один-единственный проект может сделать код чрезмерно сложным, раздутым и вечно нестабильным. Здесь напрашивается разделение между основной функциональностью (оповещение ведущего узла о проблемах) и непосредственным обнаружением конкретных проблем. API для оповещения основан на универсальных условиях и событиях. А обнаружением неполадок должны заниматься отдельные демоны — каждый в своем контейнере. Таким образом мы охватим все новые проблемы, не влияя на основной сервис Node Problem Detector. Кроме того, панель управления может содержать контроллер для автоматического исправления некоторых неполадок в узлах.



Сейчас (в Kubernetes 1.10) демоны для обнаружения проблем упакованы в двоичный файл сервиса Node Problem Detector и выполняются в качестве гоурутин¹, поэтому преимущества слабо связанной архитектуры пока недоступны.

В этом разделе мы обсудили важную тему — неполадки на отдельных узлах, которые могут помешать успешному распределению рабочей нагрузки, и то, как Node Problem Detector может помочь справиться с ними. Далее рассмотрим различные проблемные ситуации и способы их решения с помощью Heapster, централизованного журналирования, панели управления Kubernetes и Node Problem Detector.

Примеры потенциальных проблем

В крупных кластерах Kubernetes многое может пойти не так, как было задумано, и к этому нужно быть готовыми. Некоторые проблемы (в основном человеческий фактор) реально минимизировать, если жестко следовать рекомендациям и применять более строгие процессы. Но определенных неприятностей, таких как аппаратные

¹ Средство для реализации параллелизма в языке Go.

неисправности и сетевые неполадки, полностью не избежать. Иногда и людские ошибки допустимы, если приоритет — высокая скорость разработки. В этом разделе мы поговорим о разнообразных проблемах, об их обнаружении и о том, как оценить их серьезность и найти подходящее решение для исправления ситуации.

Проектирование устойчивых систем

При проектировании устойчивой системы первым делом следует перечислить потенциальные проблемные режимы, оценить вероятность возникновения каждой неполадки и ее последствия. Затем можно подумать о разных мерах предотвращения и нивелирования проблем, стратегиях ограничения потерь и управления рисками, а также о процедурах восстановления. После этого можно разработать план, в котором риски сопоставляются с профилями смягчения проблем (с учетом расходов). Такой комплексный подход имеет большое значение и должен постоянно обновляться по мере развития системы. Данный процесс необходимо подстраивать под каждую конкретную организацию. Устойчивость и восстановление после ошибок основываются на обнаружении сбоев и вероятности их устранения. В следующих подразделах описываются распространенные виды сбоев, способы их обнаружения и источники дополнительной информации.

Аппаратные сбои

Аппаратные сбои в Kubernetes можно разделить на две группы:

- ☐ узел не отвечает;
- ☐ узел отвечает.

Когда узел не отвечает, сложно сказать, с чем именно связана проблема — с сетью, конфигурацией или оборудованием. Очевидно, мы не можем провести диагностику на самом узле или извлечь с него журнальные файлы. Что же делать? Прежде всего нужно понять, отзывался ли узел с самого начала. Если он только что был добавлен в кластер, проблема, скорее всего, связана с конфигурацией. А если уже был частью кластера на протяжении какого-то времени, информацию о нем можно получить с помощью Heapster или в центральном журнале. В этом случае нужно искать признаки ухудшения производительности и ошибки, свидетельствующие о потенциальном аппаратном сбое.

Но даже если узел отвечает, он все равно способен испытывать сбои дополнительного оборудования, такого как несистемные жесткие диски или отдельные ядра. Вы можете узнать об этом из событий или оповещений о состоянии узла, которые Node Problem Detector передает ведущему серверу. Заметно также, что поды постоянно перезагружаются или задания выполняются медленнее. Все это говорит о вероятном аппаратном сбое. Еще один верный признак — то, что проблемы ограничиваются лишь одним узлом и такие стандартные меры, как перезагрузка, не помогают избавиться от данных симптомов.

Если кластер развернут в облаке, заменить потенциально неисправный узел очень просто. Вы можете вручную удалить проблемную ВМ и выделить вместо нее новую. В некоторых случаях этот процесс имеет смысл автоматизировать и применять контроллер восстановления, как в архитектуре Node Problem Detector. Он будет отслеживать проблемы или пропущенные проверки работоспособности и автоматически заменять неисправные узлы. Такой подход способен стать актуальным и в условиях частного хостинга или при использовании «голого железа». Для этого нужно всегда держать наготове пул запасных узлов. Крупномасштабные кластеры в большинстве случаев могут нормально функционировать и при снижении общей мощности. Это означает, что либо потеря нескольких узлов незначительна, либо количество узлов в системе с самого начала было немного избыточным. Таким образом, в случае неполадок у вас есть пространство для маневра.

Квоты, общие ресурсы и лимиты

Платформа Kubernetes мультиарендная. Она должна эффективно использовать свои ресурсы, однако их выделение, а также планирование работы подов основываются на системе проверок и противовесов между доступными квотами/лимитами для каждого пространства имен и запросами на получение гарантированных ресурсов со стороны подов и контейнеров. Позже мы рассмотрим это подробнее. А пока подумаем о том, что может пойти не так и как об этом узнать. Существует несколько нежелательных сценариев, с которыми реально столкнуться.

- ❑ **Нехватка ресурсов.** Если под требует определенной вычислительной мощности и памяти, а узлов с такими возможностями нет, его выполнение нельзя запланировать.
- ❑ **Неполное использование.** Под может задекларировать потребность в определенной процессорной мощности и объеме памяти (и Kubernetes ее удовлетворит), но в итоге задействовать лишь небольшую их часть. Это расточительно.
- ❑ **Неподходящая конфигурация узла.** Под, требующий много процессорного времени, но очень мало памяти, может быть развернут на узле с большим объемом памяти и занять все его вычислительные ресурсы. Таким образом, узел застопорится и не сможет обслужить другие поды, хотя память при этом будет простаивать.

Просмотр панели управления способен помочь найти неоптимальные компоненты. Чрезмерно загруженные и недостаточно используемые узлы и поды способны указывать на несоответствие квот и запрашиваемых ресурсов (рис. 3.12).

Обнаружив такое несоответствие, вы можете воспользоваться командой `describe` на уровне узла или пода. В крупномасштабных кластерах должны быть предусмотрены автоматические проверки, в ходе которых загруженность сравнивают с доступными ресурсами. Это важно, поскольку в крупных системах эти показатели способны колебаться и никто не ожидает однородной нагрузки.

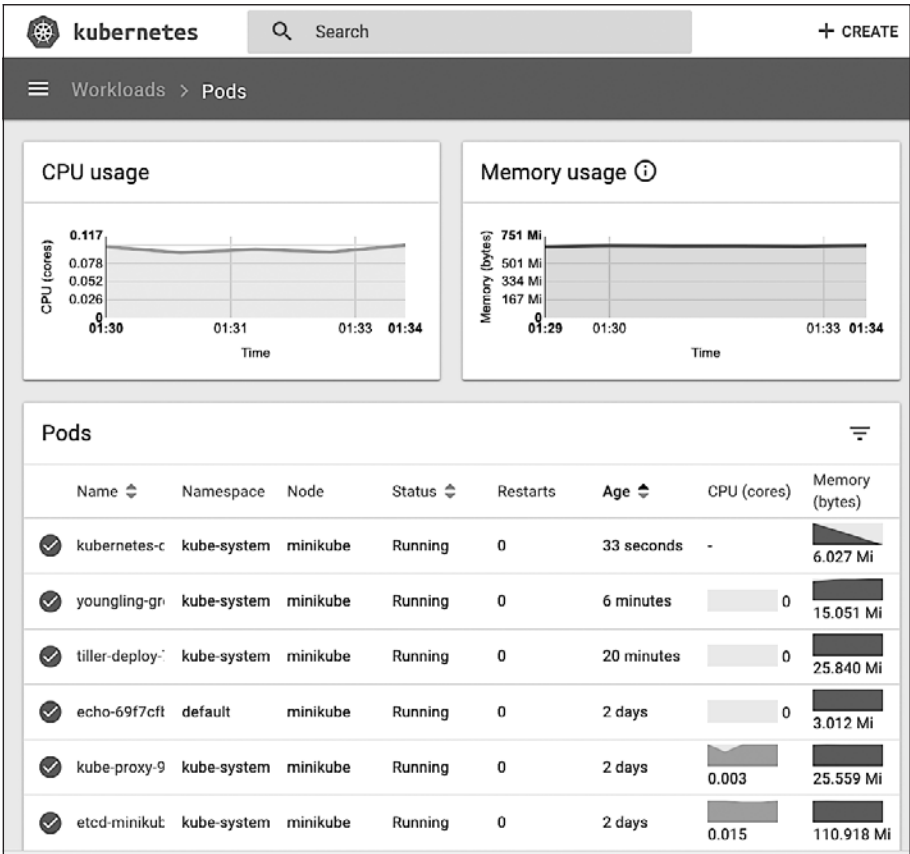


Рис. 3.12

Убедитесь в том, что понимаете требования своей системы, а ресурсы кластера находятся в допустимом диапазоне или при необходимости могут автоматически адаптироваться.

Плохая конфигурация

Плохая конфигурация — это широкое понятие. В него входит состояние кластера, настройки и аргументы командной строки контейнеров, все переменные окружения, используемые Kubernetes, вашими или любыми сторонними сервисами, а также все конфигурационные файлы. В системах с активным применением данных все связанное с конфигурацией находится в разных хранилищах. Проблемы с конфигурацией являются крайне распространенными, потому что обычно для ее тестирования нет устоявшихся методик. Часто конфигурация представлена несколькими файлами (их ищут в списке путей), один из которых используется по

умолчанию, при этом в рабочих системах и средах для разработки и тестирования применяются разные настройки.

Вы можете столкнуться с множеством разных проблем, связанных с конфигурацией кластера.

- ❑ Неподходящие метки узлов, подов или контейнеров.
- ❑ Планирование работы подов в обход контроллера репликации.
- ❑ Указание неправильных портов для сервисов.
- ❑ Некорректная карта конфигурации.

Большинство этих проблем решаются за счет подходящего процесса автоматического развертывания, но вы все равно должны хорошо разбираться в архитектуре своего кластера и взаимодействии ресурсов внутри Kubernetes.

Обычно проблемы с конфигурацией возникают после какого-нибудь изменения. Очень важно проверять состояние кластера после каждого развертывания или правки, выполненной вручную. Помочь в этом могут Heapster и панель управления. Я советую начать с сервисов: проверить их доступность, отзывчивость и функциональность. Затем можно копнуть глубже и убедиться в том, что производительность системы находится в заданных рамках.

Журнальные записи тоже содержат полезные сведения и могут навести вас на конкретные параметры конфигурации.

Соотношение затрат и производительности

Большие кластеры стоят немалых денег, особенно если они размещены в облаке. Важной частью управления крупномасштабными системами является учет расходов.

Управление расходами в облаке

Одно из основных преимуществ облака заключается в его гибкости: оно может удовлетворить меняющиеся потребности систем, которые автоматически расширяются и сжимаются, выделяя и освобождая ресурсы по мере необходимости. Платформа Kubernetes отлично вписывается в эту модель — ее всегда можно масштабировать за счет выделения новых узлов. Проблема в том, что при отсутствии надлежащих ограничений любая DoS-атака (преднамеренная, случайная или инициированная изнутри) способна привести к произвольному выделению дорогих ресурсов. Такие ситуации нужно тщательно отслеживать и пресекать в зародыше. В этом могут помочь квоты на пространства имен, но вы все равно должны уметь находить источник проблемы — это может быть внешний фактор (атака ботнета), неправильная конфигурация, неудачно выполненная внутренняя проверка или ошибка в коде для мониторинга или выделения ресурсов.

Управление расходами на «голом железе»

При использовании «голого железа» обычно можно не беспокоиться о выделении чрезмерных ресурсов, однако здесь легко упереться в потолок, если нужна дополнительная производительность, которую вы не в состоянии вовремя обеспечить. Обнаружение возросших потребностей на раннем этапе путем планирования вычислительной мощности и отслеживания производительности системы — основная задача системного администрирования. Heapster может помочь распознать долгосрочные тенденции и определить как пиковые нагрузки, так и общий рост потребности в ресурсах.

Управление расходами в гибридных кластерах

Гибридные кластеры размещаются одновременно на «голом железе» и в облаке (или на частном хостинге). Здесь нужно учитывать те же факторы. Подробно о гибридных кластерах поговорим позже.

Использование Prometheus

Heapster и стандартные средства мониторинга и журналирования, входящие в состав Kubernetes, являются отличной отправной точкой для работы. Однако в сообществе Kubernetes есть множество инноваций, среди которых и несколько альтернативных решений. Одно из самых популярных называется Prometheus. В этом разделе вы познакомитесь с целым новым миром операторов и системой Prometheus Operator, узнаете, как ее установить и задействовать для мониторинга своего кластера.

Что такое операторы

Операторы — это новый класс программного обеспечения, основой создания которого стал практический опыт разработки, управления и обслуживания приложений поверх Kubernetes. Данный термин был предложен в конце 2016 года командой CoreOS. Оператор — это контроллер отдельно взятого приложения, который расширяет API Kubernetes, позволяя создавать и конфигурировать экземпляры сложных систем с состоянием, а также управлять ими от имени пользователя Kubernetes. В нем используются базовые для Kubernetes концепции — ресурс и контроллер, но при этом он содержит сведения о домене или приложении, которые позволяют автоматизировать распространенные задачи.

Prometheus Operator

Prometheus (<https://prometheus.io>) — это набор инструментов с открытым исходным кодом для мониторинга приложений в кластере и оповещения о неполадках. Его разработчики черпали вдохновение в проекте Borgmon компании Google.

Система Prometheus рассчитана на модель назначения и планирования заданий, которая применяется в Kubernetes. В 2016 году она присоединилась к организации CNCF и получила широкое распространение. Основное ее отличие от InfluxDB состоит в том, что она использует пассивный механизм, благодаря которому кто угодно может присоединиться к конечной точке `/metrics`. Кроме того, ее язык запросов очень выразителен, но проще, чем SQL-подобный язык в InfluxDB.

В Kubernetes имеется встроенная поддержка показателей, предоставляемых Prometheus, и совместимость постепенно становится обоюдной. Все эти замечательные средства мониторинга собраны в единый пакет под названием Prometheus Operator.

Установка Prometheus с помощью kube-prometheus

Для установки Prometheus проще всего использовать пакет kube-prometheus, который основан на Prometheus Operator, информационной панели Grafana и компоненте AlertManager для управления оповещениями. Для начала клонируйте репозиторий и запустите скрипт `deploy`:

```
> git clone https://github.com/coreos/prometheus-operator.git
> cd contrib/kube-prometheus
> hack/cluster-monitoring/deploy
```

Этот скрипт создает пространство имен для мониторинга и множество служебных и вспомогательных компонентов внутри Kubernetes:

- ☐ собственно Prometheus Operator;
- ☐ `node_exporter` из состава Prometheus;
- ☐ показатели `kube-state`;
- ☐ конфигурацию Prometheus, выполняющую мониторинг всех основных компонентов и экспортных модулей Kubernetes;
- ☐ стандартный набор правил оповещения о работоспособности компонентов кластера;
- ☐ сервер Grafana, выводящий показатели кластера в графическом виде;
- ☐ трехузловой высокодоступный кластер Alertmanager.

Убедимся, что все установилось как следует:

```
> kg po -namespace=monitoring
```

NAME	READY	STATUS	RESTARTS	AGE
alertmanager-main-0	2/2	Running	0	1h
alertmanager-main-1	2/2	Running	0	1h
alertmanager-main-2	0/2	Running	0	1h
grafana-7d966ff57-rvpwk	2/2	Running	0	1h
kube-state-metrics-5dc6c89cd7-s9n4m	2/2	Running	0	1h
node-exporter-vfbhq	1/1	Running	0	1h
prometheus-k8s-0	2/2	Running	0	1h
prometheus-k8s-1	2/2	Running	0	1h
prometheus-operator-66578f9cd9-5t6xw	1/1	Running	0	1h

Как видите, компонент `alertmanager-main-2` находится в состоянии ожидания. Причиной, как мне кажется, является утилита `Minikube`, запущенная на двух ядрах. В моем случае это не вызывает никаких проблем.

Мониторинг кластера с помощью Prometheus

Установив `Prometheus Operator`, `Grafana` и `Alertmanager`, вы можете использовать их веб-интерфейсы и взаимодействовать с разными компонентами:

- ☐ веб-интерфейс `Prometheus` доступен на порте `30900`;
- ☐ веб-интерфейс `Alertmanager` доступен на порте `30903`;
- ☐ веб-интерфейс `Grafana` доступен на порте `30902`.

`Prometheus` поддерживает умопомрачительное количество показателей. На рис. 3.13 показан снимок с длительностью HTTP-запросов в микросекундах, сгруппированных по контейнерам.

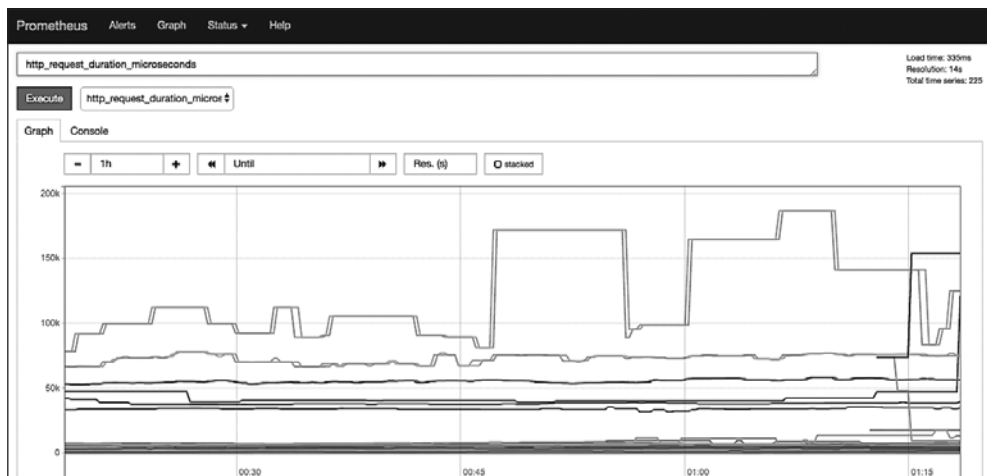


Рис. 3.13

Чтобы ограничить показатели сервиса `prometheus-k8s` квантилем 0,99, используйте следующий запрос (рис. 3.14):

```
http_request_duration_microseconds{service="prometheus-k8s", quantile="0.99"}
```

Еще один важный элемент мониторинга в `Prometheus` — `Alertmanager`. На рис. 3.15 приведен снимок веб-интерфейса, в котором можно создавать и настраивать оповещения на основе произвольных показателей.

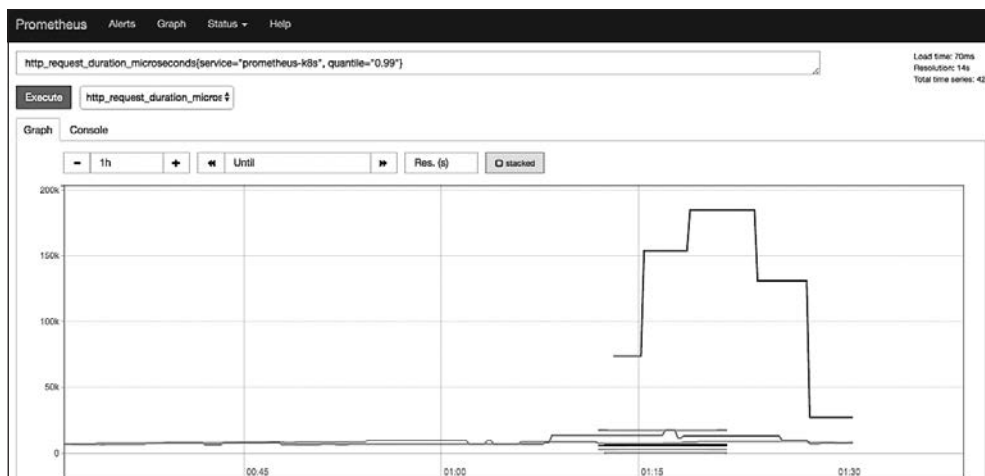


Рис. 3.14

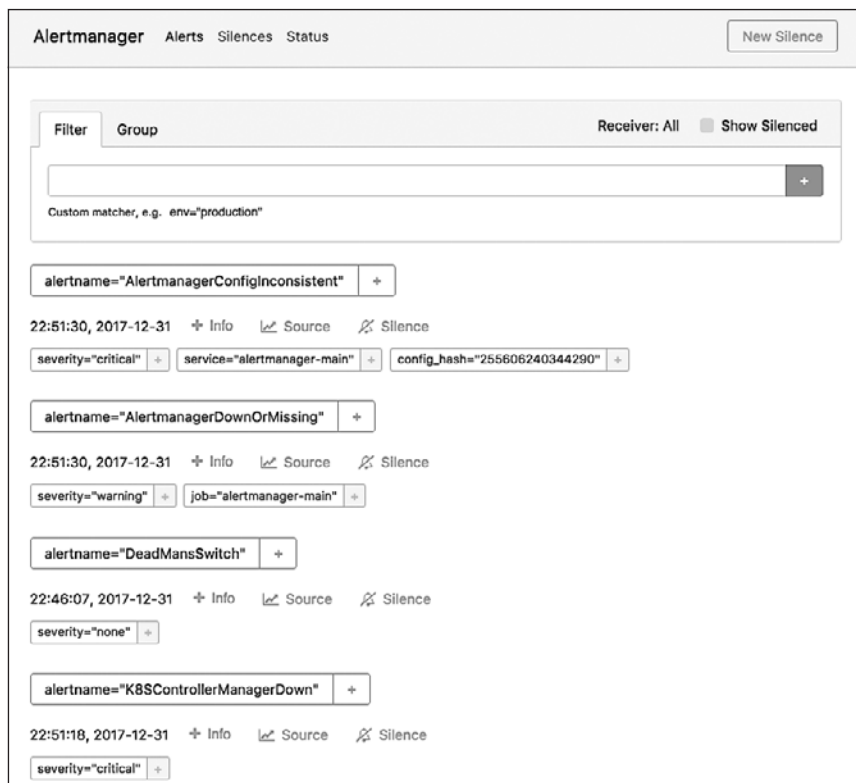


Рис. 3.15

Резюме

В данной главе мы рассмотрели мониторинг, журналирование и решение проблем. Это важнейший аспект управления любой системой, особенно такой сложной, как Kubernetes. Всякий раз, когда я отвечаю за администрирование, меньше всего мне хочется столкнуться с ситуацией, когда что-то идет не так, как предполагалось, а у меня нет надежного метода диагностики и устранения проблемы. Kubernetes предлагает обширный выбор встроенных инструментов и средств, таких как Heapster, журналирование, DaemonSets и Node Problem Detector. Но вы можете развернуть любую систему мониторинга по своему усмотрению.

В главе 4 мы обсудим высокодоступные и масштабируемые кластеры. Это, наверное, самый важный сценарий использования Kubernetes, в котором данная платформа выгодно отличается от других решений для оркестрации.

4

Высокая доступность и надежность

В предыдущей главе мы обсудили мониторинг кластера Kubernetes, обнаружение проблем на уровне узла, определение и исправление неполадок, связанных с производительностью, а также решение проблем общего характера.

В данной главе подробно рассмотрим тему высокодоступных кластеров. Это непростая область. Ни в команде Kubernetes, ни в сообществе, сложившемся вокруг этого проекта, нет единого мнения о том, как достичь идеально высокой доступности. В контексте кластеров Kubernetes этот вопрос имеет множество нюансов: необходимо обеспечить функционирование панели управления во время сбоев, сохранить состояние кластера в хранилище etcd, защитить данные системы и быстро восстановить работоспособность/производительность. В разных системах требования к надежности и доступности могут различаться. От них зависит способ реализации высокодоступного кластера.

Прочитав эту главу, вы поймете, в чем заключаются различные концепции, связанные с высокой доступностью, познакомитесь с рекомендуемыми методами ее достижения в Kubernetes и узнаете, когда их следует применять. Вы научитесь обновлять кластеры на лету, используя разные стратегии и методики, и выбирать наиболее подходящее решение с учетом производительности, расходов и степени доступности.

Концепции, связанные с высокой доступностью

Начнем путешествие в мир высокой доступности с изучения концепций и компонентов, из которых состоит надежная и высокодоступная система. Вопрос на миллион (триллион?) долларов: как соорудить надежную высокодоступную систему из ненадежных элементов? Вы неизбежно будете испытывать проблемы с отдельными элементами: оборудование ломается, сеть выходит из строя, конфигурация может быть некорректной, программное обеспечение написано с ошибками, люди тоже ошибаются. Мы должны принять это как факт и спроектировать систему так, чтобы она оставалась надежной и доступной даже при отказе отдельных компонентов. Начнем с избыточности, а затем научимся обнаруживать и быстро заменять неисправные части системы.

Избыточность

Избыточность — это основа надежных и высокодоступных систем на аппаратном уровне и в контексте хранения данных. Если вы хотите, чтобы система продолжала работать при сбое важного компонента, у вас наготове должен быть идентичный компонент. Kubernetes берет на себя поды, лишённые состояния, используя контроллеры репликации и наборы реплик. Однако для обеспечения устойчивости к сбоям состояние вашего кластера в etcd и сами ведущие компоненты должны обладать избыточностью. Кроме того, если состояние компонентов не копируется в резервное хранилище, например в облачную платформу, для предотвращения потери данных тоже необходимо обеспечить избыточность.

Горячая замена

Горячая замена заключается в замене неисправных компонентов на лету без выключения всей системы и с минимальными перебоями на стороне пользователя (а лучше их отсутствием). Если у компонента нет состояния или оно находится в избыточном хранилище, его горячая замена будет заключаться лишь в перенаправлении клиентов на новый компонент. Но если локальное состояние имеется, в том числе в памяти, эта процедура приобретает дополнительное значение. У вас есть два варианта:

- ☐ отказаться от выполнения транзакций на лету;
- ☐ поддерживать реплику в актуальном состоянии.

Первое решение простое. Большинство систем довольно устойчивы и могут пережить сбой. Неудачные клиентские запросы можно выполнить заново, и свежий компонент их обслужит.

Второе решение более сложное и хрупкое. Оно предполагает дополнительные расходы, так как каждое взаимодействие должно выполняться (и подтверждаться) в двух экземплярах. Но в некоторых частях системы такой подход может оказаться необходимым.

Выбор лидера

Выбор лидера — это распространенный шаблон проектирования в распределенных системах. Зачастую у вас имеется несколько идентичных компонентов, которые работают совместно и разделяют общую нагрузку, один из них выбирают лидером, и через него сериализуются определенные операции. Это сочетание избыточности и горячей замены. Все компоненты избыточны, и когда текущий лидер выходит из строя или становится недоступным, вместо него на лету выбирается другой.

Умная балансировка нагрузки

Под балансировкой нагрузки понимают ее распределение между несколькими компонентами для обслуживания входящих запросов. Когда какие-то компоненты выходят из строя, балансировщик первым делом должен перестать слать им запросы. Затем следует выделить новые компоненты, чтобы восстановить мощность системы и обновить балансировщик. В Kubernetes для этого предусмотрены отличные средства на основе сервисов, конечных точек и меток.

Идемпотентность

Многие виды неполадок являются временными. Это относится прежде всего к сетевым проблемам и слишком короткому времени ожидания. Компонент, который не отвечает при проверке работоспособности, считается недоступным и подлежит замене. Ранее запланированная нагрузка может быть передана другому компоненту, но прежний компонент способен оставаться функциональным и выполнять ту же работу. В итоге задача будет выполнена дважды. Этого очень сложно избежать. Чтобы работа выполнялась *строго* один раз, вам придется смириться с дополнительными расходами, ухудшением производительности, увеличением задержек и повышенной сложностью. Поскольку в большинстве систем задачи выполняются *минимум* один раз, это означает, что многократное проделывание одной и той же работы не влияет на целостность данных системы. Данное свойство называют идемпотентностью. Идемпотентные системы сохраняют свое состояние, если операция выполняется несколько раз.

Автоматическое восстановление

Когда динамическая система испытывает сбой, обычно ожидается, что она сама себя восстановит. Контроллеры репликации и наборы реплик в Kubernetes являются отличными примерами автоматического восстановления, но неисправности могут затрагивать далеко не только поды. В предыдущей главе мы обсуждали мониторинг ресурсов и обнаружение неисправностей на узлах, а также упоминали контроллер восстановления, который прекрасно вписывается в эту концепцию. Проблемы должны обнаруживаться и решаться автоматически. Квоты и лимиты помогают создать механизм сдерживающих факторов и противовесов, который не даст системе автоматического восстановления выйти из-под контроля в случае непредвиденных ситуаций, таких как DDoS-атака.

В данном разделе мы рассмотрим различные концепции, связанные с созданием надежных и высокодоступных систем. А позже применим эти знания на практике и продемонстрируем методики, рекомендованные для систем, развернутых в кластерах Kubernetes.

Рекомендуемые методики достижения высокой доступности

Построение надежных и высокодоступных распределенных систем — важная задача. В этом разделе мы познакомимся с некоторыми из рекомендуемых методик, позволяющими системам, основанным на Kubernetes, демонстрировать надежную работу и оставаться доступными при разного рода сбоях.

Создание высокодоступных кластеров

Чтобы сделать кластер Kubernetes высокодоступным, нужно добиться избыточности ведущих компонентов. Это означает, что сервис etcd должен быть развернут в виде кластера (обычно на трех или пяти узлах), а API-сервер — иметь резервные копии. Вспомогательные средства управления кластером, такие как хранилище Heapster, тоже при необходимости можно сделать избыточными. Далее представлена типичная схема надежного и высокодоступного кластера Kubernetes. Она предусматривает несколько ведущих узлов с балансировкой нагрузки, каждый из которых содержит полный набор ведущих компонентов и сервис etcd (рис. 4.1).

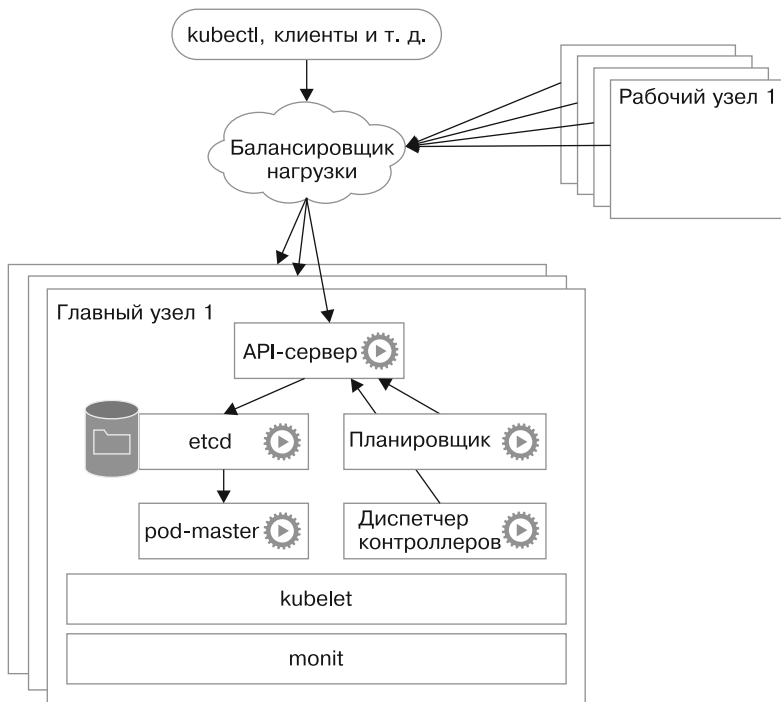


Рис. 4.1

Это не единственный вариант конфигурации высокодоступных кластеров. Например, можно развернуть отдельный кластер etcd, чтобы адаптировать узлы к рабочей нагрузке или если нужно сделать сервис etcd более избыточным по сравнению с другими ведущими узлами.

При использовании собственного хостинга к платформе Kubernetes можно применить ее собственные подходы, развернув управляющие компоненты в виде подов и наборов реплик с состоянием. Это позволит упростить достижение устойчивости и автоматическое восстановление компонентов панели управления.

Как сделать узлы надежными

Вы непременно столкнетесь с неисправностями узлов или компонентов, но многие сбои являются временными. Как минимум следует убедиться в том, что в случае сбоя демон Docker (или другая реализация CRI) и kubelet автоматически перезапускаются.

Если вы применяете CoreOS — современную ОС на основе Debian (включая Ubuntu версии 16.04 или выше) или любой другой дистрибутив с `systemd` в качестве механизма инициализации, Docker и kubelet можно легко развернуть в виде самозапускающихся демонов:

```
systemctl enable docker  
systemctl enable kubelet
```

Для других операционных систем команда Kubernetes предлагает обеспечивать высокую доступность с помощью утилиты `monit`, но вы можете выбрать любой другой монитор процессов по своему усмотрению.

Как обезопасить состояние кластера

В Kubernetes состояние кластера хранится в etcd. Кластер etcd распределен между несколькими узлами и считается суперустойчивым. Необходимо использовать эти качества при создании надежного и высокодоступного кластера.

Кластеризация etcd

Кластер etcd должен состоять минимум из трех узлов. Для повышения надежности и избыточности можно использовать любое нечетное число (пять, семь и т. д.). Количество узлов должно быть нечетным, чтобы в случае разделения сети у вас было явное большинство.

Для того чтобы можно было сформировать кластер, узлы etcd должны уметь находить друг друга. Этого можно достичь несколькими путями. Я рекомендую задействовать замечательный проект `etcd-operator` из состава CoreOS (рис. 4.2).

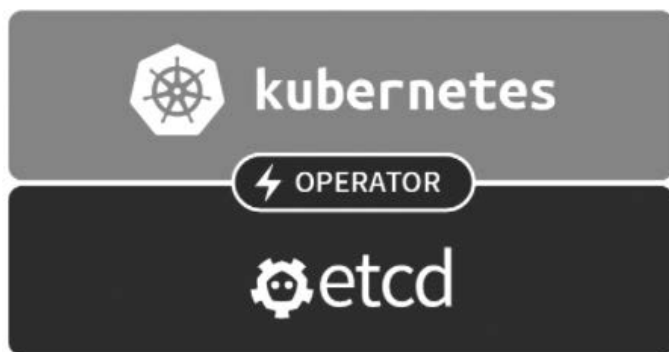


Рис. 4.2

Etcd-operator отвечает за множество сложных аспектов работы с etcd, таких как:

- ☐ создание и удаление;
- ☐ изменение размера;
- ☐ отказоустойчивость;
- ☐ плавающие обновления;
- ☐ резервное копирование и восстановление.

Установка etcd-operator

Проще всего etcd-operator можно установить с помощью Helm — диспетчера пакетов Kubernetes. Если у вас его нет, зайдите на страницу github.com/kubernetes/helm#install и следуйте приведенным там инструкциям.

Инициализируйте helm:

```
> helm init
Creating /Users/gigi.sayfan/.helm
Creating /Users/gigi.sayfan/.helm/repository
Creating /Users/gigi.sayfan/.helm/repository/cache
Creating /Users/gigi.sayfan/.helm/repository/local
Creating /Users/gigi.sayfan/.helm/plugins
Creating /Users/gigi.sayfan/.helm/starters
Creating /Users/gigi.sayfan/.helm/cache/archive
Creating /Users/gigi.sayfan/.helm/repository/repositories.yaml
Adding stable repo with URL:
https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /Users/gigi.sayfan/.helm.
Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.
Happy Helming!
```

Мы подробно обсудим Helm в главе 13, а пока воспользуемся им для установки оператора etcd. В стандартной поставке дистрибутива Minikube 0.24.1 с поддержкой Kubernetes 1.8 (хотя уже доступна версия 1.10) существуют некоторые проблемы с правами. Чтобы их обойти, придется создать определенные роли и их привязки. Далее показан файл `rbac.yaml`:

```
# Широкий доступ к кластеру (в основном для kubelet)
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: cluster-writer
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["*"]
  - nonResourceURLs: ["*"]
    verbs: ["*"]
# Полный доступ на чтение к API и ресурсам
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: cluster-reader
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["get", "list", "watch"]
  - nonResourceURLs: ["*"]
    verbs: ["*"]
# Пользователи admin, kubelet, kube-system и kube-proxy получают
# неограниченный доступ
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: cluster-write
subjects:
  - kind: User
    name: admin
  - kind: User
    name: kubelet
  - kind: ServiceAccount
    name: default
    namespace: kube-system
  - kind: User
    name: kube-proxy
roleRef:
  kind: ClusterRole
  name: cluster-writer
  apiGroup: rbac.authorization.k8s.io
```

Вы можете применить эти изменения так же, как и любой другой файл манифеста в Kubernetes:

```
kubect1 apply -f rbac.yaml
```

Наконец-то мы можем установить etcd-operator. Укажем x в качестве названия выпуска, чтобы сделать вывод более лаконичным (если хотите, используйте более выразительное название):

```
> helm install stable/etcd-operator -name x
NAME: x
LAST DEPLOYED: Sun Jan 7 19:29:17 2018
NAMESPACE: default
STATUS: DEPLOYED
RESOURCES:
→ v1beta1/ClusterRole
NAME                                AGE
x-etcd-operator-etcd-operator      1s
→ v1beta1/ClusterRoleBinding
NAME                                AGE
x-etcd-operator-etcd-backup-operator 1s
x-etcd-operator-etcd-operator        1s
x-etcd-operator-etcd-restore-operator 1s
→ v1/Service
NAME                                TYPE           CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
etcd-restore-operator              ClusterIP      10.96.236.40  <none>       19999/TCP  1s
→ v1beta1/Deployment
NAME                                DESIRED  CURRENT  UP-TO-DATE
AVAILABLE AGE
x-etcd-operator-etcd-backup-operator 1         1         1           0
1s
x-etcd-operator-etcd-operator        1         1         1           0
1s
x-etcd-operator-etcd-restore-operator 1         1         1           0
1s
→ v1/ServiceAccount
NAME                                SECRETS  AGE
x-etcd-operator-etcd-backup-operator 1         1s
x-etcd-operator-etcd-operator        1         1s
x-etcd-operator-etcd-restore-operator 1         1s
NOTES:
1. etcd-operator deployed.
   If you would like to deploy an etcd-cluster set cluster.enabled to true
   in values.yaml
Check the etcd-operator logs
  export POD=$(kubectl get pods -l app=x-etcd-operator-etcd-operator -
    namespace default -output name)
  kubectl logs $POD -namespace=default
```

Создание кластера etcd

Сохраните следующий код в файл etcd-cluster.yaml:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "etcd-cluster"
```

```
spec:
  size: 3
  version: "3.2.13"
```

Чтобы создать кластер, введите такую команду:

```
> k create -f etcd-cluster.yaml
etcdcluster "etcd-cluster" created
Let's verify the cluster pods were created properly:
> k get pods | grep etcd-cluster
etcd-cluster-0000          1/1      Running   0      4m
etcd-cluster-0001          1/1      Running   0      4m
etcd-cluster-0002          1/1      Running   0      4m
```

Проверка кластера etcd

Установив и запустив кластер etcd, проверьте его состояние и исправность с помощью утилиты etcdctl. Kubernetes позволяет запускать программы прямо внутри подов и контейнеров, используя команду `exec` (по аналогии с `exec` в Docker).

Вот как можно проверить, исправен ли кластер:

```
> k exec etcd-cluster-0000 etcdctl cluster-health
member 898a228a043c6ef0 is healthy: got healthy result from
http://etcd-cluster-0001.etcd-cluster.default.svc:2379
member 89e2f85069640541 is healthy: got healthy result from
http://etcd-cluster-0002.etcd-cluster.default.svc:2379
member 963265fbd20597c6 is healthy: got healthy result from
http://etcd-cluster-0000.etcd-cluster.default.svc:2379
cluster is healthy
```

А так — получать и устанавливать значения по ключу:

```
> k exec etcd-cluster-0000 etcdctl set test "Yeah, it works"
Yeah, it works
> k exec etcd-cluster-0000 etcdctl get test
```

Действительно, все работает!

Сохранность данных

Защита состояния и конфигурации кластера — важная задача, но еще более важно обеспечить сохранность собственных данных. Кластер всегда можно построить с нуля, если его состояние будет повреждено (хотя во время этого процесса он окажется недоступным). Но повреждение или потеря ваших данных чреват серьезными последствиями. Здесь действует все тот же закон: избыточность — наше все. Однако, в отличие от крайне динамичного состояния кластера Kubernetes, данные могут быть более статическими. Например, вы можете выполнять резервное копирование и восстановление важных данных, которые собирали на протяжении длительного времени. Текущая информация может быть утеряна, но у вас будет возможность откатить систему до предыдущей точки восстановления и тем минимизировать ущерб.

Избыточные API-серверы

API-серверы не сохраняют свое состояние — все необходимые данные они получают на лету из хранилища etcd. Это означает, что можно легко запустить несколько их экземпляров и не беспокоиться об их координации. Сделав это, разместите перед ними балансировщик нагрузки, чтобы на клиентской стороне все выглядело прозрачно.

Выбор лидера в Kubernetes

У некоторых ведущих компонентов, таких как планировщик и диспетчер контроллеров, не может быть сразу несколько активных экземпляров. Представьте, что разные планировщики пытаются развернуть один и тот же под на разных узлах или поместить несколько его копий на один узел — это ввергло бы систему в хаос. Чтобы добиться высокого уровня масштабируемости кластера Kubernetes, данные компоненты следует запускать в режиме выбора лидера. Это означает, что при наличии нескольких экземпляров только один из них может быть активным, если он выйдет из строя, вместо него будет выбран другой.

В Kubernetes такой режим устанавливается с помощью флага `leader-elect`. Планировщик и диспетчер контроллеров можно развернуть в виде подов, скопировав их манифесты в каталог `/etc/kubernetes/manifests`.

Далее представлен фрагмент из манифеста планировщика, в котором используется этот флаг:

```
command:
- /bin/sh
- -c
- /usr/local/bin/kube-scheduler -master=127.0.0.1:8080 -v=2 -leaderelect=
true 1>>/var/log/kube-scheduler.log
2>&1
```

А это аналогичный фрагмент из манифеста диспетчера контроллеров:

```
- command:
- /bin/sh
- -c
- /usr/local/bin/kube-controller-manager -master=127.0.0.1:8080 -clustername=
e2e-test-bburns
--cluster-cidr=10.245.0.0/16 -allocate-node-cidrs=true -cloudprovider=
gce -service-account-private-key-file=/srv/kubernetes/server.key
--v=2 -leader-elect=true 1>>/var/log/kube-controller-manager.log 2>&1
image: gcr.io/google_containers/kube-controllermanager:
fda24638d51a48baa13c35337fcd4793
```

Заметьте, что, в отличие от других подов, эти компоненты не могут автоматически перезапускаться средствами Kubernetes, поскольку именно они отвечают за

перезапуск вышедших из строя экземпляров пода. У вас уже должна быть запущена готовая замена.

Выбор лидера в приложении. Выбор лидера может быть полезным и внутри вашего приложения, но его чрезвычайно сложно реализовать. К счастью, в этом вам способна помочь система Kubernetes. На такой случай предусмотрена задокументированная процедура с применением контейнера `leader-elect` от компании Google. Ее основная идея заключается в использовании конечных точек Kubernetes в сочетании с полями `ResourceVersion` и `Annotations`. Подключив этот контейнер к поду своего приложения, вы получите удобные средства для выбора лидера.

Запустим контейнер `leader-elect` с тремя подами и проведем выборы под названием `election`:

```
> kubectl run leader-elect --image=gcr.io/google_containers/leadelector:0.5 --replicas=3 -- --election=election -http=0.0.0.0:4040
```

Через какое-то время вы увидите в своем кластере три новых пода с именами вида `leader-elector-xxx`:

```
> kubectl get pods | grep elect
leader-elector-57746fd798-7s886          1/1      Running    0
39s
leader-elector-57746fd798-d94zx          1/1      Running    0
39s
leader-elector-57746fd798-xcljl          1/1      Running    0
39s
```

Хорошо, но который из них ведущий? Получим конечные точки `election`:

```
> kubectl get endpoints election -o json
{
  "apiVersion": "v1",
  "kind": "Endpoints",
  "metadata": {
    "annotations": {
      "control-plane.alpha.kubernetes.io/leader":
        "{\"holderIdentity\":\"leader-elector-57746fd798-xcljl\",
        \"leaseDurationSeconds\":10,\"acquireTime\":\"2018-01-08T04:16:40Z\",
        \"renewTime\":\"2018-01-08T04:18:26Z\",\"leaderTransitions\":0}"
    },
    "creationTimestamp": "2018-01-08T04:16:40Z",
    "name": "election",
    "namespace": "default",
    "resourceVersion": "1090942",
    "selfLink": "/api/v1/namespaces/default/endpoints/election",
    "uid": "ba42f436-f42a-11e7-abf8-080027c94384"
  },
  "subsets": null
}
```

Если хорошо присмотреться, ответ можно найти в разделе `metadata.annotations`. Чтобы упростить поиск, советую использовать фантастическую утилиту `jq` для работы с JSON (<http://stedolan.github.io/jq/>). Она крайне полезна при анализе вывода Kubernetes API или `kubectl`:

```
> kubectl get endpoints election -o json | jq -r .metadata.annotations[] |
jq .holderIdentity
"leader-elect-57746fd798-xcljl"
```

Чтобы убедиться в том, что выбор лидера работает, удалим текущего лидера и посмотрим, появится ли у него замена:

```
> kubectl delete pod leader-elect-916043122-10wjj
pod "leader-elect-57746fd798-xcljl" deleted
```

И вот наш новый лидер:

```
> kubectl get endpoints election -o json | jq -r .metadata.annotations[] |
jq .holderIdentity
"leader-elect-57746fd798-d94zx"
```

Лидера можно найти и по HTTP, так как контейнер `leader-elect` предоставляет соответствующую информацию через локальный веб-сервер на порте **4040** (спрятанный за прокси):

```
> kubectl proxy
In a separate console:
> curl
http://localhost:8001/api/v1/proxy/namespaces/default/pods/leader-elect-57746fd798-d94zx:4040/ | jq .name
"leader-elect-57746fd798-d94zx"
```

Локальный веб-сервер позволяет `leader-elect` подключиться к контейнеру вашего основного приложения в рамках одного пода. Оба этих контейнера находятся в одной локальной сети, поэтому можете обратиться по адресу `http://localhost:4040`, чтобы узнать имя текущего лидера. Из всех контейнеров с вашим приложением активным будет только тот, который находится в одном поде с выбранным лидером, контейнеры в остальных подах будут простаивать. Любой полученный запрос они перенаправят лидеру, хотя путем хитрых манипуляций с балансировщиком нагрузки можно сделать так, чтобы лидер получал все запросы напрямую.

Высокая доступность в тестовой среде

Высокая доступность имеет большое значение. Если вам никак не удастся ее достичь, это означает, что у вас появилась новая бизнес-задача. И прежде, чем разворачивать свой кластер в промышленной среде, вам необходимо убедиться в его надежности и высокой доступности (разве что вы работаете в Netflix, где тестирование происходит в реальных условиях). Кроме того, любое изменение, внесенное

в кластер, теоретически способно ухудшить его высокую доступность, не влияя на другие функции. Все сводится к следующему правилу, которое действует везде: если вы что-то не протестировали, считайте, что оно не работает.

Итак, мы установили, что необходимо проверять надежность и высокую доступность. Лучше всего это делать в тестовой среде, которая наиболее точно эмулирует промышленные условия. Да, это может быть накладно, но есть несколько способов снизить расходы.

- ❑ **Временная высокодоступная среда.** Создавайте кластер только на время тестирования его высокой доступности.
- ❑ **Насыщенность.** Заранее организуйте интересующие вас потоки событий и сценарии, подготавливайте ввод и симулируйте ситуации одну за другой.
- ❑ **Сочетайте проверку высокой доступности и производительности со стресс-тестированием.** Проверив высокую доступность и производительность, перезагрузите систему и посмотрите, как ее конфигурация с этим справится.

Тестирование высокой доступности

Тестирование высокой доступности требует планирования и глубоких знаний о системе. Цель всех проверок — выявление недостатков архитектуры и/или реализации. Покрытие тестами должно быть хорошим, чтобы в случае их прохождения вы были уверены в том, что поведение системы будет предсказуемым.

В контексте надежности и высокой доступности это означает, что вам нужно каким-то образом сломать свою систему и проследить за тем, как она восстановится. Данную задачу можно разбить на несколько пунктов.

- ❑ Составить исчерпывающий список вероятных неполадок и их потенциальных сочетаний.
- ❑ Предусмотреть для каждого случая четкую реакцию.
- ❑ Определить способ провокации неполадок.
- ❑ Найти способ отслеживания реакции системы.

Каждый из пунктов важен. По моему опыту, лучше всего делать все это постепенно, установив относительно небольшое количество общих категорий сбоев и способов реагирования на них. Попытка составить всеобъемлющий постоянно меняющийся список низкоуровневых неполадок — не самая лучшая идея.

Например, общая категория сбоев называется «узел не отвечает», а стандартной реакцией на нее может быть перезагрузка узла. Спровоцировать такой сбой можно, остановив ВМ с узлом (если это ВМ). Со стороны это должно выглядеть, будто узел недоступен, но приемочное тестирование должно показать, что система работает в нормальном режиме. Существует множество других аспектов, которые следует протестировать, например, были ли записаны в журнал сведения об отказе, получили ли оповещения соответствующие люди, обновились ли разные статистические данные и отчеты.

Стоит отметить, что иногда проблему нельзя решить за один подход. Так, если узел перестал отвечать из-за аппаратных проблем, перезагрузка не поможет. В этом случае должно активизироваться дополнительное решение, например запуск и конфигурация новой ВМ, к которой затем подключится подходящий узел. Здесь можно пренебречь универсальностью и создать тесты для определенных типов подов, размещенных на узле (etcd, ведущая, рабочая, база данных и мониторинг).

Если вам нужно обеспечить высокое качество системы, приготовьтесь к тому, что на конфигурацию тестовых сред и отдельных тестов будет уходить даже больше времени, чем на развертывание в промышленных условиях.

Последний и наиболее важный совет: пытайтесь выбирать как можно более тонкие решения. Это означает, что в идеале у готовой системы не должно быть механизмов мониторинга, которые позволяли бы отключать целые ее части или снижать ее производительность на время проверки, так как это делает ее более уязвимой к атакам и ошибкам конфигурации. В идеале управление тестовой средой не должно требовать изменения кода или конфигурации, которые будут развернуты в промышленных условиях. Обычно в Kubernetes довольно легко внедрить поды и контейнеры с дополнительными тестовыми функциями, которые взаимодействуют с системными компонентами на этапе тестирования, но никогда не развертываются в рабочей среде.

В данном разделе мы рассмотрели средства для получения надежного и высокодоступного кластера, включая etcd, API-сервер, планировщик и диспетчер контроллеров. Вы познакомились с методиками, которые рекомендуется использовать для защиты как самого кластера, так и ваших данных, уделив при этом особое внимание проблемам, связанным с тестовыми средами и тестированием.

Обновление кластера на лету

Одна из самых сложных и рискованных процедур, связанных с обслуживанием кластера Kubernetes, заключается в горячих обновлениях. Часто взаимодействие между различными версиями разных частей системы сложно предсказать, но во многих ситуациях без этого не обойтись. Крупные кластеры с множеством пользователей не могут простаивать во время обслуживания. Со сложностью лучше всего бороться методом «разделяй и властвуй». Здесь очень к месту приходится архитектура микросервисов. Ваша система никогда не обновляется целиком. Вместо этого постоянным обновлениям подлежат лишь отдельные наборы связанных между собой микросервисов, при изменении API изменяются и их клиенты. Хорошо продуманное обновление предусматривает сохранение обратной совместимости до тех пор, пока не обновятся все клиенты, затем на протяжении нескольких версий старые API помечаются как неактуальные.

В этом разделе поговорим о разных стратегиях обновления кластеров, таких как плавающие (rolling) и сине-зеленые (blue-green) обновления. Вы также узнаете, в каких ситуациях можно вносить ломающие изменения, отказываясь от обратной совместимости. Затем рассмотрим важные темы — миграцию данных и их структуры.

Плавающие обновления

Плавающее обновление подразумевает постепенный перевод компонентов на следующую версию. Это означает, что в вашем кластере будут одновременно присутствовать как действующие, так и новые компоненты. Здесь возможны два варианта:

- ☐ новые компоненты имеют обратную совместимость;
- ☐ новые компоненты не имеют обратной совместимости.

В первом случае обновление должно быть простым. В ранних версиях Kubernetes с плавающими обновлениями следовало обходиться крайне осторожно, используя метки и постепенно меняя количество реплик для новых и старых версий (хотя у контроллеров репликации для этого предусмотрена удобная команда `kubectl rolling-update`). Но в Kubernetes 1.2 появился новый ресурс развертывания, который значительно упрощает эту процедуру и поддерживает наборы реплик. Он имеет следующие возможности:

- ☐ выполнение на стороне сервера (продолжает работать даже после отключения вашего компьютера);
- ☐ ведение версий;
- ☐ множественные параллельные выкатывания;
- ☐ обновляющие развертывания;
- ☐ агрегация состояния всех подов;
- ☐ откат изменений;
- ☐ развертывание самых свежих изменений;
- ☐ множественные стратегии (плавающие обновления по умолчанию).

Далее приведен пример манифеста для развертывания трех подов с NGINX:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Тип этого ресурса — `Deployment` (развертывание), а его имя, `nginx-deployment`, можно использовать позже для обращения к развертыванию (например, обновления

или отката изменений). Самая важная часть — шаблон пода, — конечно же, находится в разделе `spec`. Реплики определяют, сколько подов будет в кластере, а шаблон содержит конфигурацию для каждого контейнера (в данном случае лишь для одного).

Чтобы начать плавающее обновление, создайте ресурс развертывания:

```
$ kubectl create -f nginx-deployment.yaml --record
```

Позже можно узнать о состоянии развертывания с помощью следующей команды:

```
$ kubectl rollout status deployment/nginx-deployment
```

Сложное развертывание. Ресурс развертывания отлично подходит в ситуациях, когда нужно обновить один под, но часто таких подов оказывается несколько и иногда их версии зависят друг от друга. В таком случае вам, возможно, придется отказаться от плавающих обновлений или ввести новый временный слой совместимости. Представьте, что сервис А зависит от сервиса Б. В какой-то момент в сервис Б внесли серьезные изменения. Поды версии 1 сервиса А больше не могут взаимодействовать с подами версии 2 сервиса Б. Кроме того, поддержка подами версии 2 сервиса Б одновременно новых и старых API нежелательна с точки зрения надежности и управления изменениями. Для решения этой проблемы можно создать сервис-адаптер, который реализует API сервиса Б версии 1. Он размещается между А и Б, транслируя запросы и ответы между версиями. Это усложняет процесс развертывания и требует дополнительных действий, но сервисы А и Б остаются простыми. Вы можете откатывать обновления несовместимых версий, а промежуточный слой исчезнет, когда все поды сервисов А и Б перейдут на версию 2.

Сине-зеленые обновления

Плавающие обновления повышают доступность системы, но иногда их выкатывание оказывается слишком сложным или требует значительных усилий, замедляя разработку более важных проектов. В таких ситуациях отличной альтернативой могут послужить сине-зеленые обновления. Они подразумевают предварительную подготовку полной копии вашей промышленной среды с новой версией. Таким образом, вы получаете две копии — старую (синюю) и новую (зеленую). На самом деле цвета не играют никакой роли — главное, чтобы у вас были две автономные промышленные среды. Вначале активна синяя среда, она обслуживает все запросы. Тем временем вы можете начать выполнять все тесты в зеленой среде и, удовлетворившись результатом, сделать ее активной. Если что-то пойдет не так, легко откатиться обратно — просто переключитесь с зеленой среды на синюю. Я деликатно проигнорировал тему хранилища и состояния, находящегося в памяти. Мгновенное переключение между средами подразумевает, что их компоненты не обладают состоянием и используют общий слой хранения данных.

Обновление хранилища или ломающие изменения API, доступных внешним клиентам, требуют принятия дополнительных мер. Так, если синяя и зеленая

среды имеют отдельные хранилища, перед переключением вам, скорее всего, придется направлять все входящие запросы каждой из них и синхронизировать ранее сохраненные данные.

Управление изменениями в контрактах данных

Контракты данных описывают то, как данные должны быть организованы. Это общий термин для метаданных с описанием структуры. Типичный пример — структура базы данных. К ним относятся также информация, передающаяся по сети, форматы файлов и даже содержимое строковых аргументов и ответов. Любой файл конфигурации обладает как форматом (JSON, YAML, TOML, XML, INI или другим, нестандартным), так и некой внутренней структурой, которая описывает допустимые виды иерархий, ключей, значений и типов данных. Контракты данных бывают общими и конкретными. В любом случае работать с ними нужно внимательно, иначе вы получите ошибки выполнения, когда код, занимающийся чтением, разбором или проверкой, столкнется с незнакомой структурой данных.

Миграция данных

Миграция данных имеет большое значение. Многие современные системы работают с данными, объем которых измеряется терабайтами, петабайтами и т. д. В будущем количество собираемой и обрабатываемой информации будет только расти, и скорость этого процесса уже опережает темпы развития аппаратных решений. Проще говоря, если данных много, их перенесение может занять определенное время. В компании, где я прежде работал, мне довелось наблюдать за миграцией 100 Тбайт из кластера Cassandra одной устаревшей системы в другой аналогичный кластер.

У второго кластера Cassandra была иная структура, и кластер Kubernetes обращался к нему непрерывно. Эта задача была очень сложной, и, когда возникали неотложные проблемы, ее выполнение постоянно откладывалось. По истечении изначально поставленных сроков старая и новая системы еще долго работали параллельно.

Средств для разделения данных и передачи их обоим кластерам было достаточно, но в какой-то момент мы столкнулись с проблемами масштабирования новой системы и должны были их решить, прежде чем двигаться дальше. Старые данные имели большое значение, но доступ к ним не нужно было обслуживать на том же уровне, что и свежую информацию. Поэтому мы занялись еще одним проектом — переносом старых данных в более дешевое хранилище. Естественно, это означало, что клиентские библиотеки и сервисы должны уметь обращаться к обоим хранилищам и объединять полученные результаты. Работая с большими объемами данных, ничего нельзя принимать на веру. Вы будете сталкиваться с проблемами масштабирования инструментов, инфраструктуры, сторонних зависимостей и собственных процессов. Большой масштаб — часто не только количественная, но и качественная характеристика. Не надейтесь, что все пройдет гладко. Это намного сложнее, чем файлы, расположенные в точке А, просто скопировать в точку Б.

Устаревание API

Устаревание API бывает внутренним и внешним. Внутренние API применяют компоненты, полностью подконтрольные вашей команде или организации. Вы можете быть уверены, что все, кто ими пользуется, перейдут на новую версию в кратчайшие сроки. С внешними API работают пользователи или сервисы за пределами вашей непосредственной сферы влияния. В некоторых неоднозначных ситуациях даже с внутренними интерфейсами приходится обращаться как с внешними — например, если вы работаете в огромной компании, такой как Google. Если повезет, все ваши API будут использоваться автоматически обновляемыми приложениями или через веб-страницу, которую вы контролируете. В таких случаях программный интерфейс практически скрыт от внешнего мира и вам даже не нужно его публиковать.

Если у вашего API много пользователей (или несколько очень важных клиентов), его устаревание следует тщательно контролировать. Если интерфейс устарел, вы должны заставить пользователей обновиться, иначе старая версия никуда не денется.

Есть несколько способов смягчить связанные с этим неудобства.

- ❑ Обходитесь без устаревания. Расширяйте имеющийся API или продолжайте поддерживать предыдущий. Иногда это довольно легко сделать, хотя без дополнительного тестирования не обойтись.
- ❑ Предоставляйте своей целевой аудитории клиентские библиотеки для всех значимых языков программирования. Это хороший вариант. Вы сможете вносить произвольные изменения в исходный API, не вызывая неудобств для пользователей (главное, чтобы интерфейс языка программирования был стабильным).
- ❑ Если вам нужно объявить интерфейс устаревшим, предоставьте пользователям достаточное количество времени для перехода на новую версию и по мере возможности посодействуйте им в этом (допустим, создайте руководство по обновлению с примерами). Они это оценят.

Производительность, расходы и компромиссы архитектуры крупных кластеров

В предыдущем разделе мы рассмотрели обновление кластеров на лету. Вы познакомились с различными методиками и их реализацией в Kubernetes. Мы также обсудили такие проблемы, как ломающие изменения, изменение контрактов данных, миграция данных и устаревание API. Этот раздел посвящен разным видам крупных кластеров и их конфигурации с разной степенью надежности и доступности. При проектировании кластера необходимо иметь представление об имеющихся вариантах и делать выбор в зависимости от потребностей своей организации.

Мы пройдемся по различным требованиям к доступности, начиная с отсутствия всяких гарантий и заканчивая таким желанным нулевым временем простоя, и поговорим о том, что это означает с точки зрения производительности и расходов.

Требования к доступности

У различных систем и даже подсистем могут быть совершенно разные требования к надежности и доступности. Например, биллинговые приложения всегда имеют высокий приоритет, поскольку в случае их недоступности вы не сможете принимать оплату. Но если у такого приложения иногда будет отказывать страница, на которой можно обжаловать выставленный счет, это будет вполне приемлемо с точки зрения бизнеса.

Отсутствие гарантий

Иногда такой подход называют *best effort* («по мере возможности»). Если все работает — прекрасно! Если нет — что ж, бывает. Такая степень надежности и доступности уместна для постоянно меняющихся внутренних компонентов — их просто невыгодно делать устойчивыми. Это может подойти и для сервисов, выпускаемых в виде бета-версии.

С точки зрения разработчиков, это отличный подход. Вы можете быстро продвигаться вперед, ломая совместимость. Вам не нужно беспокоиться о последствиях, получать одобрение и проходить через вереницу тестов. По своей производительности сервисы с отсутствием гарантий способны выигрывать у более надежных аналогов, так как могут пропускать некоторые затратные этапы, например проверку запросов, сохранение промежуточных результатов и репликацию данных. Однако многие надежные сервисы хорошо оптимизированы, а вспомогательное оборудование подобрано специально под их нагрузку. Отсутствие гарантий обычно снижает расходы, так как не нужно обеспечивать избыточность (разве что администраторы пренебрегут планированием нагрузки и просто выделяют лишние ресурсы).

В контексте Kubernetes встает важный вопрос: у всех ли сервисов, которые предоставляет кластер, отсутствуют гарантии? Если это так, тогда сам кластер не должен быть высокодоступным. Вы, скорее всего, обойдетесь одним ведущим узлом с единственным экземпляром etcd, возможно, даже не придется развертывать Heapster или другое решение для мониторинга.

Периоды обслуживания

Процедуры обслуживания системы, такие как применение заплаток безопасности, обновление программного обеспечения, разбиение журнальных файлов и очистка баз данных, обычно планируются на определенное время. В такие периоды система (или подсистема) становится недоступной. Пользователей часто уведомляют о запланированном простое. Преимущество данного подхода в том, что вам не нужно беспокоиться о последствиях, которые ваши действия оказывают на текущие запросы, поступающие в систему. Это может чрезвычайно упростить процесс. Системные администраторы обожают периоды обслуживания не меньше, чем разработчики — отсутствие гарантий.

В Kubernetes на время простоя все входящие запросы можно перенаправлять на веб-страницу (или возвращать JSON-ответ) с уведомлением о техническом обслуживании. Для этого используется балансировщик нагрузки.

Но в большинстве случаев платформа Kubernetes оказывается достаточно гибкой для обслуживания на лету. В экстремальных ситуациях, таких как обновление Kubernetes или перевод etcd с версии 2 на версию 3, вы можете запланировать период обслуживания. Еще одной альтернативой являются сине-зеленые обновления. Однако с увеличением масштабов кластера они становятся все более затратными, так как придется дублировать систему целиком, что не только дорого, но и чревато проблемами, связанными с недостаточно большими квотами.

Быстрое восстановление

Еще один важный аспект высокодоступных кластеров — быстрое восстановление. Рано или поздно у вас что-то сломается. В этот момент начнут тикать часы, отсчитывающие время простоя. Как скоро вы сможете вернуться к нормальной работе?

Иногда от вас ничего не зависит. Допустим, если у вашего облачного провайдера перебои в работе, а вы не реализовали многокластерный режим, о котором поговорим позже, вам придется сидеть и ждать, пока проблемы не будут устранены. Но чаще всего неполадки возникают из-за недавнего развертывания. Некоторые проблемы могут быть связаны с временем или даже календарем. Помните ошибку с високосным годом, которая 29 февраля 2012 года привела к сбою Microsoft Azure?

Конечно, идеальный пример быстрого восстановления — сине-зеленое развертывание. Это когда на время поиска проблемы делается откат на предыдущую версию.

Неплохим решением могут быть и плавающие обновления: если вы обнаружите проблему на ранней стадии, большинство подов не успеют перейти на новую версию.

Устранение проблем, связанных с данными, может занять много времени, даже если у вас есть актуальные резервные копии и процедура восстановления действительно работает (не забывайте регулярно ее проверять).

В отдельных ситуациях способны помочь инструменты вроде Neptio Ark, которые позволяют делать резервные снимки кластера: если что-то идет не так и вы не знаете, как это исправить, можете просто восстановить снимок.

Нулевое время простоя

Наконец-то мы добрались до систем с нулевым временем простоя. На самом деле таким систем не бывает. Все на свете ломается, и программного обеспечения это касается в первую очередь. Иногда неполадки достаточно серьезные для того, чтобы вывести из строя всю систему или отдельные ее сервисы. О нулевом времени простоя следует думать в контексте распределенной архитектуры с отсутствием гарантий. Вы закладываете в свою систему достаточную степень избыточности

и механизмы, которые позволят ей оставаться доступной во время решения вероятных проблем. И не забывайте, что, даже если простой исключен согласно вашим бизнес-требованиям, это не касается отдельно взятых компонентов.

План достижения нулевого времени простоя выглядит так.

- ❑ **Избыточность на всех уровнях.** Это необходимое условие. У вашей архитектуры не может быть единой точки отказа, иначе ее сбой сделает недоступной всю систему.
- ❑ **Автоматическая горячая замена неисправных компонентов.** Избыточность хороша только в том случае, если резервные компоненты готовы сразу же заменить собой исходные части системы, вышедшие из строя. Некоторые из них могут разделять нагрузку (например, веб-серверы без сохранения состояния), не требуя от вас каких-то особенных действий. Но в случае с такими компонентами Kubernetes, как планировщик и диспетчер контроллеров, нужно обеспечить выбор лидера, чтобы кластер продолжал работать как прежде.
- ❑ **Активный мониторинг и множество оповещений для раннего обнаружения проблем.** Даже при тщательном проектировании реально что-то пропустить или сделать опрометчивое предположение. Такие неявные проблемы имеют свойство накапливаться, но если быть внимательными, их можно выявить до того, как они перерастут в системный сбой. Представьте, что у вас есть механизм для очистки журнальных файлов при заполнении диска на 90 %, но по какой-то причине он не работает. Если установить оповещение об использовании 95 % места на диске, вы сможете обнаружить эту проблему и предотвратить отказ системы.
- ❑ **Тщательное тестирование перед развертыванием в промышленных условиях.** Комплексные тесты зарекомендовали себя как надежный способ улучшения качества. Полноценное тестирование таких сложных и крупных проектов, как кластер Kubernetes, управляющий огромной распределенной системой, — непростая, но необходимая задача. Что нужно проверять? Все! Именно так, чтобы добиться нулевого времени простоя, вам придется тестировать как само приложение, так и инфраструктуру. Стопроцентное прохождение модульных тестов может быть хорошим началом, но не гарантирует, что приложение будет работать как положено при развертывании на промышленном кластере Kubernetes. Конечно, в идеале тестирование следует проводить в реальных условиях, применяя сине-зеленое развертывание или идентичный кластер. Вместо этого можно использовать тестовую среду, наиболее похожую на оригинал. Далее перечислены тесты, которые необходимо выполнить:
 - модульные тесты;
 - приемочные тесты;
 - тесты производительности;
 - стресс-тесты;
 - тесты отката изменений;
 - тесты восстановления данных;
 - тесты на проникновение.

Каждый из них должен быть исчерпывающим, поскольку любая непроверенная деталь может сломаться. Звучит безумно? Отлично. Добиться идеальной устойчивости крупномасштабных систем чрезвычайно сложно. Не зря в Microsoft, Google, Amazon, Facebook и других больших компаниях над инфраструктурой, администрированием и обеспечением работоспособности систем трудятся десятки и сотни тысяч разработчиков (в совокупности).

- ❑ **Хранение сырых данных.** Во многих системах данные являются самым важным активом. Вы всегда сможете восстановить свою систему при повреждении или потере данных, если будете сохранять их еще до обработки. Это вряд ли поможет добиться нулевого времени простоя, так как восстановление будет далеко не мгновенным, но при этом вы не потеряете никакой информации, что часто еще важнее. Недостаток такого подхода заключается в том, что сырые данные обычно занимают намного больше места, чем обработанные. Но вы можете выделить для них более дешевое хранилище.
- ❑ **Видимость отсутствия отказов как крайняя мера.** Итак, какая-то часть системы все же отказала. Но вы по-прежнему можете обеспечивать какой-то уровень услуг. Вероятно, данные будут немного устаревшими или пользователи получат доступ к другим частям системы — это не самый лучший вариант, но формально система останется доступной.

Производительность и согласованность данных

При разработке и администрировании распределенных систем всегда следует помнить о теореме CAP (consistency, availability, partition tolerance — согласованность, доступность и устойчивость к разделению), которая гласит, что из этих трех свойств в лучшем случае можно добиться лишь двух. Поскольку на практике любая система способна столкнуться с разделением сети, вам остается выбор между CP и AP. CP означает, что для поддержания согласованности система будет недоступной на время разделения сети. Согласно AP разделение сети может нарушить только согласованность системы, но не ее доступность. Например, чтение из разных разделов может дать разные результаты, поскольку один из них мог не получить запрос на запись. В этом разделе мы сосредоточимся на высокодоступных системах, то есть AP. Для достижения высокой доступности придется пожертвовать согласованностью, но это вовсе не значит, что наши приложения будут содержать поврежденные или произвольные данные. Ключевым здесь является то, что в конечном счете данные согласуются — возможно, система будет немного запаздывать и выдавать слегка устаревшую информацию, но рано или поздно вы получите то, что ожидаете. Такой подход делает потенциально возможным существенное улучшение производительности.

Так, если вы часто (скажем, каждую секунду) обновляете какой-то существенный показатель, но передаете его лишь раз в минуту, это позволит уменьшить расход трафика в 60 раз, а средний возраст обновлений при этом будет составлять лишь 30 с. Это имеет огромное значение. Теперь количество пользователей или запросов, которые может обслуживать ваша система, выросло в 60 раз.

Резюме

В данной главе мы рассмотрели надежные и высокодоступные кластеры. Это, наверное, идеальная ниша для Kubernetes. И хотя эта платформа способна оркестрировать мелкие кластеры с небольшим количеством контейнеров, ее настоящая ценность заключается в масштабировании систем. Для этого вам нужно проверенное решение с подходящими инструментами и устоявшимися методиками.

Теперь вы получили четкое представление о надежности и высокой доступности в распределенных системах. Познакомились с лучшими методиками для создания надежных и высокодоступных кластеров Kubernetes, исследовали нюансы их обновления на лету и научились идти на определенные компромиссы, когда речь о производительности и расходах.

В следующей главе мы уделим внимание важной теме — безопасности в Kubernetes. Вы узнаете о проблемах и рисках, связанных с этой областью, а также подробно изучите такие составляющие, как пространства имен, служебные учетные записи, контроль доступа, аутентификация, авторизация и шифрование.

5 Конфигурация безопасности, ограничений и учетных записей в Kubernetes

В главе 4 мы рассмотрели надежные и высокодоступные кластеры Kubernetes, связанные с ними основные понятия, рекомендуемые методики, обновления на лету и многие компромиссы, относящиеся к производительности и расходам.

В этой главе исследуем тему безопасности. Кластер Kubernetes — сложная система с компонентами, взаимодействующими между собой на разных уровнях. Изоляция и разграничение этих уровней очень значимы в процессе работы с важными приложениями. Чтобы обезопасить систему и обеспечить надлежащий доступ к ресурсам, возможностям и данным, сначала нужно понять, с какими уникальными вызовами сталкивается универсальная платформа для оркестрации при выполнении произвольных задач. Далее воспользуемся преимуществами различных механизмов безопасности, изоляции и контроля доступа, чтобы защитить сам кластер, приложения, которые на нем выполняются, и их данные. Мы обсудим разные устоявшиеся методики и покажем, когда стоит использовать тот или иной механизм.

По прочтении этой главы вы будете четко понимать, какие проблемы безопасности решает Kubernetes. А также научитесь делать свои системы устойчивыми к разного рода атакам, выстраивать многоуровневую защиту и даже конфигурировать безопасные мультиарендные кластеры, предоставляя своим пользователям полную изоляцию и контроль за их частью системы.

Проблемы безопасности, стоящие перед Kubernetes

Kubernetes — чрезвычайно гибкая платформа, которая применяет универсальные методы для управления очень низкоуровневыми ресурсами. Ее можно развертывать в разных операционных системах и облачных средах, используя виртуальные машины и «голое железо». Kubernetes выполняет задачи, реализованные в средах, взаимодействие с которыми происходит через четко определенные интерфейсы, но без понимания деталей реализации. Платформа Kubernetes управляет важными аспектами — сетью, DNS и выделением ресурсов, делая это от имени или по поручению приложений, о которых ей ничего неизвестно. Это требует наличия хороших

механизмов безопасности и возможностей, которые позволяют администраторам защитить свои приложения и уберечься от распространенных ошибок.

В данном разделе мы обсудим проблемы безопасности, которые могут возникнуть на нескольких уровнях и в разных компонентах кластера Kubernetes: узлах, сетях, образах, подах и контейнерах. Важный аспект здесь — многоуровневая защита, которая позволяет нивелировать атаки на других уровнях и ограничить охват и ущерб от проникновения. Чтобы ее реализовать, первым делом нужно распознать вызовы, которые стоят перед каждым из уровней.

Потенциальные уязвимости узлов

На узлах размещаются среды выполнения. Если злоумышленник получит доступ к узлу, это может стать серьезной угрозой. В его подчинении окажутся как минимум сам компьютер и все задачи, которые на нем выполняются. Но это еще не самый плохой вариант. На узле запущена утилита kubelet, которая общается с API-сервером. Умелый взломщик может подменить kubelet модифицированной версией и тем самым избежать обнаружения, продолжая взаимодействовать с API-сервером в нормальном режиме. При этом он сможет выполнять собственные задачи вместо запланированных, собирать общие сведения о кластере и нарушать работу API-сервера или системы целиком, рассылая злонамеренные сообщения. Возможно еще более глубокое проникновение, если узел имеет доступ к общим ресурсам и конфиденциальным данным. При взломе узла угрозу несут не только вероятные последствия, но и то, что его сложно обнаружить постфактум.

Узлы могут быть скомпрометированы и на аппаратном уровне. Это относится скорее к физическим устройствам, которые можно включить в кластер Kubernetes.

Еще один вектор атаки — утечка ресурсов. Представьте, что ваши узлы стали частью ботнета, который выполняет на них собственные задачи и отбирает у кластера процессорное время и память. Опасность здесь таится в том, что Kubernetes и ваша инфраструктура могут автоматически масштабироваться и выделять дополнительные ресурсы.

Есть проблемы и с установкой инструментов для отладки и выявления неисправностей, а также с изменением конфигурации вне автоматического развертывания. Такие операции обычно не тестируются, и если оставить их без присмотра, они могут не только ухудшить производительность, но и привести к куда более губительным последствиям. В лучшем случае это может ослабить устойчивость к атакам.

Обеспечение безопасности — это своего рода лотерея. Вам необходимо знать, какие части системы подвергаются атакам. Перечислим потенциальные проблемы, связанные с узлами.

- ❑ Злоумышленник получает контроль над узлом.
- ❑ Злоумышленник подменяет kubelet.
- ❑ Злоумышленник завладевает узлом с ведущими компонентами — API-сервером, планировщиком и диспетчером контроллеров.

- ☐ Злоумышленник получает физический доступ к узлу.
- ☐ Злоумышленник направляет ресурсы кластера на сторонние задачи.
- ☐ Нанесен непреднамеренный ущерб в результате установки инструментов для отладки и выявления проблем или изменения конфигурации.

Любой значимый кластер Kubernetes охватывает как минимум одну сеть. С сетевой конфигурацией связано множество проблем. Вы должны понимать все тонкости взаимодействия своих системных компонентов. Какие из них должны общаться между собой? Какие сетевые протоколы они используют? Какие порты? Какими данными обмениваются? Каким образом кластер соединен с внешним миром?

Мы имеем дело со сложной цепочкой портов и возможностей/сервисов:

- ☐ от контейнера к узлу;
- ☐ от узла к узлу в рамках внутренней сети;
- ☐ от узла вовне.

Использование оверлейных сетей (их мы обсудим в главе 10) может помочь выстроить многоуровневую защиту: даже если злоумышленники получают доступ к Docker-контейнеру, они будут изолированы и не смогут добраться до исходной сетевой инфраструктуры.

Большую опасность таит также обнаружение компонентов. У вас есть несколько вариантов, таких как DNS, отдельные службы обнаружения и балансировщики нагрузки. У каждого из них есть свои плюсы и минусы, и их применение в конкретной ситуации требует тщательного планирования и глубокого понимания.

Важно, чтобы контейнеры могли находить друг друга и обмениваться информацией.

Вам нужно решить, какие ресурсы и конечные точки будут публично доступны. Затем вы должны подобрать подходящий метод аутентификации пользователей/сервисов и авторизовать их для работы с этими ресурсами.

Конфиденциальные данные могут шифроваться при поступлении в кластер и отправке из него, а иногда и во время хранения. Это требует управления и безопасного обмена ключами, что является одной из сложнейших задач в области безопасности.

Если ваш кластер имеет общую сетевую инфраструктуру с другими кластерами Kubernetes или сторонними процессами, нужно серьезно задуматься об их изоляции и разделении.

Существуют разные рецепты решения этих проблем, но ингредиенты все те же: сетевые политики, правила брандмауэра и *программно-определяемые сети* (software-defined networking, SDN). Особо сложными случаями являются кластеры, размещенные внутри организации и основанные на «голом железе». Давайте подытожим. Нам нужно:

- ☐ разработать схему сетевого соединения;
- ☐ выбрать компоненты, протоколы и порты;
- ☐ предусмотреть динамическое обнаружение;

- ❑ подумать о публичном и приватном доступе;
- ❑ настроить аутентификацию и авторизацию;
- ❑ написать правила для брандмауэра;
- ❑ выработать сетевую политику;
- ❑ позаботиться об управлении и обмене ключами.

С одной стороны, вам необходимо облегчить поиск и взаимодействие между собой по сети контейнеров, пользователей и сервисов. С другой — ограничить доступ и предотвратить атаки по сети или на саму сеть.

Многие из этих вызовов не имеют прямого отношения к Kubernetes. Однако данная универсальная платформа занимается управлением ключевой инфраструктурой и отвечает за низкоуровневые аспекты сети, поэтому необходимо выработать динамические и гибкие решения, которые помогут интегрировать в нее особенности разных систем.

Потенциальные уязвимости образов

Контейнеры, работающие на платформе Kubernetes, должны быть совместимы с одной из сред выполнения, которые в ней поддерживаются. Kubernetes не имеет никакого представления о том, чем эти контейнеры занимаются (помимо собираемых показателей). Можно применять к ним квоты и ограничивать их доступ к другим участкам сети с помощью сетевых политик. Но так или иначе контейнеру нужно обращаться к ресурсам узла, другим узлам в сети, распределенному хранилищу и внешним сервисам. То, как он себя ведет, определяется его образом. Проблемные образы можно разделить на две категории:

- ❑ вредоносные;
- ❑ уязвимые.

Вредоносные образы содержат код или конфигурацию, созданные злоумышленником для нанесения ущерба или сбора информации. Вредоносный код может быть внедрен в ваш процесс подготовки образов, включая любые используемые репозитории. Вы также можете установить сторонний образ, модифицированный злоумышленником и содержащий вредоносный код.

Уязвимыми могут оказаться ваши собственные или сторонние образы. Они позволяют злоумышленнику завладеть запущенным контейнером или причинить какой-то вред, например внедрить свой код на определенном этапе.

Сложно сказать, какая из категорий хуже. По большому счету, они равнозначны, так как позволяют захватить контроль над контейнером. Размер потенциального ущерба определяется другими уровнями защиты (не забывайте: наша защита многоуровневая) и ограничениями, которые вы установили для контейнера. Минимизация опасности скомпрометированных контейнеров — крайне сложная задача. Быстро развивающиеся компании, использующие микросервисы, могут ежедневно генерировать множество образов. И проверить их все тоже непросто.

Образы Docker, к примеру, состоят из разных слоев. Базовый образ, содержащий операционную систему, может в любой момент оказаться в опасности при обнаружении новой уязвимости. Более того, вредоносный код может попасть в сторонние базовые образы, которым вы доверяете и которыми пользуетесь, никак не контролируя (распространенная ситуация).

Суммируем потенциальные проблемы с образами.

- ❑ Kubernetes не знает, чем занимаются образы.
- ❑ Платформа Kubernetes обязана предоставить доступ к конфиденциальным ресурсам для заданной функции.
- ❑ Процесс подготовки и доставки образов, включая их репозитории, сложно обезопасить.
- ❑ Скорость разработки и развертывания новых образов может сказываться на том, как тщательно проверяются изменения.
- ❑ Базовые образы, содержащие ОС, могут легко устареть и стать уязвимыми.
- ❑ Часто базовые образы находятся вне вашего контроля, и в них может внедриться вредоносный код.
- ❑ Интеграция статических анализаторов образов, таких как CoreOS Clair, может серьезно помочь.

Потенциальные проблемы с конфигурацией и развертыванием

Кластерами Kubernetes управляют удаленно. В любой момент их состояние обусловлено различными манифестами и политиками. Если злоумышленник получит доступ к компьютеру, который имеет административный контроль над кластером, он сможет нанести ущерб, например собрать информацию, внедрить зараженные образы, ослабить безопасность и подделать журнальные файлы. И как всегда, не меньший вред способны причинить программные и человеческие ошибки — они могут повлиять на важные меры безопасности и оставить кластер открытым для атаки. В наши дни администраторы часто управляют кластером удаленно, находясь с ноутбуком дома или в кафе. От открытия входа в систему их отделяет лишь одна команда `kubectl`.

Еще раз перечислим потенциальные вызовы.

- ❑ Kubernetes управляют удаленно.
- ❑ Злоумышленник с удаленным административным доступом может получить полный контроль над кластером.
- ❑ Обычно конфигурацию и развертывание сложнее протестировать, чем код.
- ❑ Работающие удаленно или просто находящиеся вне офиса сотрудники рискуют случайно дать злоумышленнику доступ к своим ноутбукам или телефонам с административными привилегиями.

Потенциальные уязвимости подов и контейнеров

В Kubernetes поды являются единицей работы и содержат один или несколько контейнеров. Под — это лишь абстракция для группировки и развертывания, а на практике контейнеры, развернутые в одном поде, взаимодействуют между собой напрямую. Все контейнеры имеют общую локальную сеть и часто подключены к томам узла. Такая простая интеграция между контейнерами в одном поде может привести к открытию доступа к отдельным частям узла для всех контейнеров сразу. И если у вас есть один плохой контейнер (вредоносный или уязвимый), он может открыть дорогу для серьезных атак на другие контейнеры в поде, что впоследствии позволит получить контроль над самим узлом. Такого рода угрозу несут ведущие дополнения, которые часто находятся на одном компьютере с ведущими компонентами, особенно учитывая то, что многие из них экспериментальные. То же самое относится к набору демонов, которые запускают поды на каждом узле.

Для многоконтейнерных подов характерны следующие проблемы.

- ❑ Контейнеры в одном поде имеют общую локальную сеть.
- ❑ Иногда контейнеры в одном поде имеют общий доступ к томам в файловой системе узла.
- ❑ Плохие контейнеры могут заразить своих соседей по поду.
- ❑ Плохие контейнеры могут нести дополнительные угрозы, если они размещены рядом с компонентами, имеющими доступ к важным ресурсам узла.
- ❑ Экспериментальные дополнения, размещенные рядом с ведущими компонентами, могут быть нестабильными и менее безопасными.

Потенциальные организационные и культурные проблемы

Безопасность часто влияет на производительность. Это нормальный компромисс, и вам не стоит об этом беспокоиться. Раньше, когда разработчики и администраторы были разделены, этот конфликт улаживался на организационном уровне. Разработчики стремились добиться большей продуктивности и воспринимали требования к безопасности как издержки ведения бизнеса. Администраторы контролировали промышленную среду и отвечали за доступ и процедуры безопасности. В какой-то момент эти две области сошлись — так и появилось понятие DevOps (developers and operations — разработка и администрирование). В наши дни скорость разработки часто ставится во главу угла. Когда-то такие концепции, как непрерывное развертывание (многократное развертывание в течение одного дня без вмешательства человека), были неслыханными в большинстве организаций.

Платформа Kubernetes была создана специально для нового мира облачных приложений. Однако в ее основе лежит опыт компании Google, которая уделяет

множество времени и привлекает отличных специалистов для планирования процессов и подбора инструментов, что позволяет сочетать частое развертывание с высокой безопасностью. В организациях поменьше добиться такого сочетания зачастую крайне непросто, из-за чего может пострадать безопасность.

Организации, использующие Kubernetes, сталкиваются со следующими вызовами.

- ❑ Разработчики, которые занимаются администрированием Kubernetes, могут меньше ориентироваться на безопасность.
- ❑ Скорость разработки может считаться более важной, чем безопасность.
- ❑ В ходе непрерывного развертывания можно упустить часть проблем с безопасностью, которые в итоге дойдут до промышленной среды.
- ❑ У сотрудников небольших организаций может не хватить знаний и опыта для обеспечения надлежащей безопасности в кластерах Kubernetes.

В этом разделе мы рассмотрели множество вызовов, с которыми вам предстоит столкнуться при построении безопасного кластера. Большинство из них не имеют прямого отношения к Kubernetes, однако использование данной платформы означает, что значительная часть вашей системы универсальна и ей ничего не известно о назначении системы. Подобное может усложнить организацию безопасности. Связанные с этим проблемы обнаруживаются на разных уровнях:

- ❑ узлов;
- ❑ сети;
- ❑ образов;
- ❑ конфигурации и развертывания;
- ❑ подов и контейнеров;
- ❑ вызовов, связанных с организацией рабочего процесса.

В следующем разделе мы рассмотрим средства для решения некоторых из этих проблем, предоставляемые Kubernetes. Часто такие решения должны находиться на уровне системы. Важно понимать, что самого факта использования механизмов безопасности Kubernetes недостаточно.

Как закаляется Hardening

В предыдущем разделе перечислено множество разных проблем, с которыми сталкиваются разработчики и администраторы при развертывании и обслуживании кластеров Kubernetes. Здесь же мы сосредоточимся на аспектах архитектуры, механизмах и особенностях Kubernetes, которые должны послужить ответом на некоторые из этих вызовов. Вы сможете добиться довольно высокого уровня безопасности за счет таких механизмов, как служебные учетные записи, сетевые политики, аутентификация, авторизация, контроль доступа, AppArmor и объекты `secret`.

Не забывайте, что кластер Kubernetes — лишь часть большой системы, состоящей из других программных компонентов, людей и рабочих процессов. Kubernetes не может решить все ваши проблемы. Вы всегда должны держать в уме общие принципы безопасности, такие как многоуровневая защита, понятие «необходимое знание» и правило минимальных привилегий. Помимо этого, записывайте в журнал все, что может пригодиться в случае взлома, и подготовьте оповещения для раннего обнаружения отклонений в состоянии системы. Они могут оказаться признаком как ошибки, так и атаки. В любом случае, чтобы прореагировать, вы должны о них знать.

Служебные учетные записи в Kubernetes

У Kubernetes есть два вида учетных записей — внешние, с помощью которых к кластеру подключаются живые люди (например, через утилиту `kubectl`), и служебные.

Обычные пользователи являются глобальными и могут обращаться к разным пространствам имен внутри кластера. Служебные учетные записи ограничены одним пространством имен, что обеспечивает их изоляцию. Это важно, поскольку каждый раз, когда API-сервер получает запрос от поды, его полномочия действительны только в его собственном пространстве имен.

Kubernetes управляет служебными учетными записями от имени подов. Каждому поду при создании назначается служебная запись, которая будет идентифицировать все ее процессы при обращении к API-серверу. У каждой служебной записи есть набор полномочий, хранящихся на секретном томе. У каждого пространства имен есть стандартный служебный пользователь с именем `default`. Он назначается любому поду при создании (если только вы не указали другую служебную учетную запись).

Вы можете добавлять новые служебные учетные записи. Создайте файл с именем `custom-service-account.yaml` и следующим содержимым:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-service-account
```

Now type the following:

```
kubectl create -f custom-service-account.yaml
```

That will result in the following output:

```
serviceaccount "custom-service-account" created
```

Here is the service account listed alongside the default service account:

```
> kubectl get serviceaccounts
```

NAME	SECRETS	AGE
custom-service-account	1	3m
default	1	29d



Как можно заметить, для новой учетной записи автоматически был создан объект `secret`.

Чтобы получить больше подробностей, введите:

```
> kubectl get serviceAccounts/custom-service-account -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2018-01-15T18:24:40Z
  name: custom-service-account
  namespace: default
  resourceVersion: "1974321"
  selfLink: /api/v1/namespaces/default/serviceaccounts/custom-service-account
  uid: 59bc3515-fa21-11e7-beab-080027c94384
  secrets:
  - name: custom-service-account-token-w2v7v
```

Чтобы просмотреть сам объект `secret`, включая файл `ca.crt`, и токен, введите такую команду:

```
kubectl get secrets/custom-service-account-token-w2v7v -o yaml
```

Как Kubernetes управляет служебными учетными записями

У API-сервера есть специальный компонент, который занимается контролем доступа для служебных учетных записей. В момент создания пода он проверяет, имеет ли она собственную служебную запись и, если да, существует ли эта запись на самом деле. Если таковой не указано, поду назначается служебная учетная запись по умолчанию.

Этот компонент следит также за тем, чтобы у пода было полномочие `ImagePullSecrets`, необходимое для получения образов из удаленного реестра. Если у пода нет собственных объектов `secret`, он использует полномочие `ImagePullSecrets` своей служебной учетной записи.

В конце к поду подключается том `/var/run/secrets/kubernetes.io/service-account` с токеном для доступа к API и объектом `volumeSource`. Данный токен создается и добавляется в объект `secret` другим компонентом под названием «контроллер токенов», который делает это в момент создания служебной учетной записи. Кроме того, он добавляет/удаляет токены в момент добавления в служебную учетную запись объектов `secret` или их удаления из нее.

Контроллер служебных учетных записей гарантирует наличие стандартного пользователя во всех пространствах имен.

Доступ к API-серверу

При обращении к API необходимо пройти через этапы аутентификации, авторизации и контроля доступа. Запрос может быть отклонен на любом из них. Каждый этап представляет собой цепочку связанных между собой дополнений. Это проиллюстрировано на рис. 5.1.

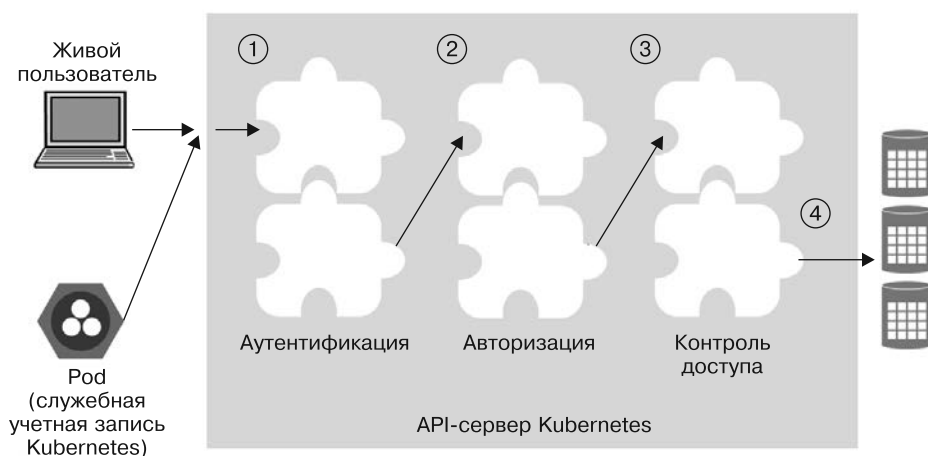


Рис. 5.1

Аутентификация пользователей

При создании нового кластера первым делом создаются клиентские сертификат и ключ. С их помощью утилита `kubectl` и API-сервер аутентифицируются при работе друг с другом по TLS на порте 443 (зашифрованное HTTPS-соединение). Вы можете найти свои клиентские ключ и сертификат в файле `.kube/config`:

```
> cat ~/.kube/config | grep client
client-certificate: /Users/gigi.sayfan/.minikube/client.crt
client-key: /Users/gigi.sayfan/.minikube/client.key
```



Если к кластеру должны обращаться другие пользователи, создателю нужно передать им клиентские сертификат и ключ безопасным способом.

Но это лишь установление базового уровня доверия с API-сервером Kubernetes. Вы все еще не аутентифицированы. Существуют различные модули аутентификации, которые могут проверять запросы на наличие дополнительных сертификатов, паролей, токенов-носителей и JWT-токенов (для служебных учетных записей). Большинство запросов предназначено для аутентифицированных пользователей (с обычными или служебными учетными записями), хотя некоторые могут быть анонимными. Если запрос не сможет пройти процесс аутентификации целиком, он будет отклонен с HTTP-кодом 401 («ошибка авторизации», что не совсем точно).

Выбором стратегий аутентификации занимается администратор кластера. Для этого он указывает API-серверу разные аргументы командной строки:

❑ `--client-ca-file=<filename>` — для клиентских сертификатов x509, заданных в файле;

- ❑ `--token-auth-file=<filename>` — для токенов-носителей, заданных в файле;
- ❑ `--basic-auth-file=<filename>` — для логина и пароля, заданных в файле;
- ❑ `--experimental-bootstrap-token-auth` — для компоновочных токенов, используемых утилитой `kubeadm`.

Служебные учетные записи применяют дополнение для аутентификации, которое загружается автоматически. Администратор может указать два дополнительных ключа:

- ❑ `--service-account-key-file=<filename>` — ключ для подписи токенов-носителей, зашифрованный методом PEM; если его не указать, будет использоваться приватный TLS-ключ API-сервера;
- ❑ `--service-account-lookup` — делает так, что токены отзываются при удалении их из API-сервера.

Существует несколько дополнительных методов — OpenID Connect, Webhooks, Keystone (служба идентификации в OpenStack) и прокси-сервер с аутентификацией. Основная идея их применения состоит в том, что этап аутентификации является расширяемым и может поддерживать любые механизмы.

После прохождения через цепочку аутентификационных дополнений запросу будут присвоены следующие атрибуты (на основе предоставленных полномочий):

- ❑ удобное для восприятия *имя пользователя*;
- ❑ *уникальный идентификатор (UID)* — постоянный, в отличие от имени пользователя;
- ❑ список *групп*, к которым принадлежит пользователь;
- ❑ *дополнительные поля* (строковые пары «ключ — значение»).

Аутентификатору ничего не известно о том, какими правами обладает пользователь. Он лишь связывает набор полномочий с набором идентификационных атрибутов. Проверкой того, может ли пользователь выполнить тот или иной запрос, занимаются авторизаторы. Аутентификация считается успешной, если хотя бы один из авторизаторов примет предоставленные полномочия. Этапы аутентификации выполняются в произвольном порядке.

Перевоплощение. Пользователи с надлежащей авторизацией могут действовать от чужого имени. Например, администратор может заняться отладкой от лица другого пользователя с меньшими привилегиями. Для этого в API-запросе нужно передать заголовки перевоплощения:

- ❑ `Impersonate-User` — пользователь, от имени которого выполняется запрос;
- ❑ `Impersonate-Group` — группа или группы, от имени которых выполняется запрос. Этот заголовок необязательный и требует наличия `Impersonate-User`;
- ❑ `Impersonate-Extra` (дополнительное имя) — это динамический заголовок, который связывает с пользователем дополнительные поля. Он необязательный и требует наличия `Impersonate-User`.

В `kubectl` нужно указать параметры `--as` и `--as-group`.

Авторизация запросов

Закончив с аутентификацией, пользователь приступает к авторизации. В Kubernetes для этого предусмотрены универсальные методы. Цепочка модулей авторизации принимает запрос, содержащий такую информацию, как аутентифицированное имя пользователя и действие (`list`, `get`, `watch`, `create` и т. д.). В отличие от аутентификации, авторизация выполняется в полном объеме при каждом запросе. Если запрос не сможет пройти проверку хотя бы в одном из модулей или не получит однозначного ответа, он будет отклонен с HTTP-кодом 403 (запрещено). Запрос продолжит свой путь, если был принят хотя бы одним модулем и при этом никакой другой модуль его не отклонил.

Выбором дополнений для авторизации занимается администратор кластера. Для этого он перечисляет их названия через запятую в аргументе командной строки `--authorization-mode`.

Поддерживаются следующие режимы:

- ❑ `--authorization-mode=AlwaysDeny` — отклоняет все запросы, подходит для тестирования;
- ❑ `--authorization-mode=AlwaysAllow` — разрешает все запросы, подходит на случай, если вам не нужна авторизация;
- ❑ `--authorization-mode=ABAC` — включает политику локальной пользовательской авторизации с помощью файла. *ABAC* расшифровывается как *attribute-based access control* (разграничение доступа на основе атрибутов);
- ❑ `--authorization-mode=RBAC` — ролевой механизм, в котором хранением и применением политик авторизации занимается API-сервер Kubernetes. *RBAC* расшифровывается как *role-based access control* (управление доступом на основе ролей);
- ❑ `--authorization-mode=Node` — это специальный режим, предназначенный для авторизации API-запросов со стороны kubelets;
- ❑ `--authorization-mode=Webhook` — позволяет выполнять авторизацию через удаленный сервис с помощью REST.

Можете добавить собственное дополнение для авторизации, реализовав простой интерфейс на языке Go:

```
type Authorizer interface {
    Authorize(a Attributes) (authorized bool, reason string, err error)
}
```

Входящий аргумент `Attributes` — тоже интерфейс. Он предоставляет все сведения, необходимые для принятия решения относительно авторизации:

```
type Attributes interface {
    GetUser() user.Info
    GetVerb() string
    IsReadOnly() bool
    GetNamespace() string
    GetResource() string
}
```

```

GetSubresource() string
GetName() string
GetAPIGroup() string
GetAPIVersion() string
IsResourceRequest() bool
GetPath() string
}

```

Использование дополнений для контроля доступа

Итак, запрос был аутентифицирован и авторизован, но перед его выполнением нужно сделать еще один шаг. Он должен пройти через вереницу дополнений для контроля доступа. По аналогии с авторизацией запрос считается неудачным, если он был отклонен хотя бы одним контроллером.

Контроллеры доступа — прекрасная концепция. Их смысл в том, что у кластера могут быть глобальные причины для отклонения запроса. И без таких контроллеров о них должны были бы знать все модули авторизации. Контроль доступа позволяет реализовать эту логику лишь один раз. Кроме того, контроллерам позволено модифицировать запросы. Их можно запускать в режиме проверки или изменения. Как обычно, выбором контроллеров доступа занимается администратор, для этого предусмотрен аргумент командной строки `--admission-control`. В качестве значения указывается список дополнений, разделенных запятыми. Далее перечислены дополнения, которые рекомендуется использовать в Kubernetes версии 1.9 или выше (порядок имеет значение):

```

--admissioncontrol=
NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,
DefaultStorageClass,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,
ResourceQuota,DefaultTolerationSeconds

```

Рассмотрим некоторые из доступных дополнений (этот список постоянно пополняется):

- ❑ `AlwaysAdmit` — безусловный пропуск (не совсем понятно, зачем оно нужно);
- ❑ `AlwaysDeny` — отклоняет все подряд (подходит для тестирования);
- ❑ `AlwaysPullImages` — всегда позволяет загружать новые образы (подходит для мультиарендных кластеров, гарантирует, что приватные образы не используются подами, у которых нет полномочий для их загрузки);
- ❑ `DefaultStorageClass` — добавляет стандартный класс хранения данных к запросам типа `PersistentVolumeClaim` (в которых не указано подобного класса);
- ❑ `DefaultTolerationSeconds` — устанавливает стандартный уровень терпимости для ограничений (если он еще не установлен): `notready:NoExecute` и `notreachable:NoExecute`;
- ❑ `DenyEscalatingExec` — запрещает команды `exec` и `attach` для подов с доступом к узлу и повышенными привилегиями. Это относится к узлам, которые имеют доступ к пространствам имен `IPC` и `PID`;

- ❑ **EventRateLimit** — ограничивает количество событий, поступающих в API-сервер (появилось в Kubernetes 1.9);
- ❑ **ExtendedResourceToleration** — сочетает ограничения для узлов со специальными ресурсами, такими как GPU и FPGA, при этом подам, которые запрашивают эти ресурсы, назначается определенный уровень терпимости. В итоге узел с избыточными ресурсами будет выделен для подов с подходящим уровнем терпимости;
- ❑ **ImagePolicyWebhook** — это сложное дополнение подключается к внешнему серверу, чтобы решить, следует ли отклонить запрос на основе его образа;
- ❑ **Initializers** — устанавливает отложенные инициализаторы, редактируя метаданные ресурса, который должен быть создан (основано на **InitializerConfiguration**);
- ❑ **InitialResources** (экспериментальное) — назначает вычислительные ресурсы и лимиты; если их не указать, будут задействованы данные о нагрузке, записанные ранее;
- ❑ **LimitPodHardAntiAffinity** — отказывает любому поду, который не использует ключ топологии непринадлежности `kubernetes.io/hostname` в `requiredDuringSchedulingRequiredDuringExecution`;
- ❑ **LimitRanger** — отклоняет запросы, нарушающие лимиты на ресурсы;
- ❑ **MutatingAdmissionWebhook** — вызывает по порядку зарегистрированные изменяющие процедуры Webhook, способные модифицировать указанный объект. Стоит отметить, что разные процедуры могут нивелировать изменения, вносимые друг другом;
- ❑ **NamespaceLifecycle** — отклоняет запросы по созданию объектов в пространствах имен, которые находятся в процессе удаления или которых не существует;
- ❑ **ResourceQuota** — отклоняет запросы, которые нарушают квоты на ресурсы, действующие в пространстве имен;
- ❑ **ServiceAccount** — это автоматизация для служебных учетных записей;
- ❑ **ValidatingAdmissionWebhook** — этот контроллер доступа вызывает любую процедуру Webhook, которая соответствует запросу. Подходящие процедуры вызываются одновременно. Запрос завершится неудачей, если хотя бы одна из них его отклонит.

Как видите, дополнения для контроля доступа имеют широкие возможности. Они поддерживают политики, действующие в рамках пространств имен, и подтверждают действительность запросов — в основном с точки зрения управления ресурсами. Благодаря этому дополнения для авторизации могут сосредоточиться на допустимых операциях. **ImagePolicyWebhook** — это метод проверки образов, что само по себе важная задача. **Initializers** делает возможным динамический контроль доступа, позволяя вам развертывать собственные контроллеры, не встраивая их в Kubernetes. Существуют также внешние процедуры Webhook для контроля доступа. Они подходят для таких задач, как семантическая проверка ресурсов (например, чтобы гарантировать, что все поды имеют стандартный набор меток).

Разбиение проверки входящих запросов на разные стадии (аутентификация, авторизация и контроль доступа), каждая со своими дополнениями, значительно упрощает понимание и применение этого сложного процесса.

Защита подов

Безопасность подов очень важна, поскольку Kubernetes планирует и выполняет их запуск. Для обеспечения безопасности подов и контейнеров существует несколько независимых механизмов. Вместе они формируют многоуровневую защиту: даже если злоумышленник (или программная ошибка) обойдет один механизм, он будет заблокирован другим.

Использование частных репозиториев с образами

Этот подход даст вам уверенность в том, что кластер будет загружать только те образы, которые вы предварительно проверили. Он также улучшает управление обновлениями. Вы можете настроить `$HOME/.dockercfg` или `$HOME/.docker/config.json` на каждом узле, хотя многие облачные провайдеры этого не позволяют, поскольку узлы в них выделяются автоматически.

ImagePullSecrets

Данный подход рекомендуется для кластеров на облачных платформах. Его суть в том, что полномочия для реестра предоставляются самим подом, поэтому неважно, на каком узле он развернут. Это решает проблему с файлом `.dockercfg` на уровне узла.

Сначала следует создать объект `secret` для полномочий:

```
> kubectl create secret the-registry-secret
--docker-server=<docker registry server>
--docker-username=<username>
--docker-password=<password>
--docker-email=<email>
secret "docker-registry-secret" created.
```

При необходимости объекты `secret` можно создавать для нескольких реестров или для разных пользователей в рамках одного реестра. Все экземпляры `ImagePullSecrets` будут объединены с помощью `kubelet`.

Однако поды могут обращаться только к объектам `secret` внутри собственного пространства имен, поэтому вы должны поместить эти объекты во все пространства имен, в которых хотите запускать свой под.

Определив объект `secret`, можете добавить его в спецификацию пода, которую затем сможете запустить. В результате под будет использовать полномочия из объекта `secret` для загрузки образов из заданного репозитория:

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: cool-pod
namespace: the-namespace
spec:
  containers:
    - name: cool-container
      image: cool/app:v1
  imagePullSecrets:
    - name: the-registry-secret

```

Задание контекста безопасности

Контекст безопасности — это набор параметров уровня операционной системы, таких как UID, GID, возможности и роли SELinux. Эти параметры применяются на уровне контейнера как часть его защищенного содержимого. Вы можете задать контекст безопасности, который будет действовать для всех контейнеров внутри определенного пода. Контекст этого уровня может использовать и свои параметры безопасности (в частности, `fsGroup` и `seLinuxOptions`) к томам.

Далее приведен пример контекста безопасности уровня пода:

```

apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    ...
  securityContext:
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: "s0:c123,c456"

```

Контекст безопасности уровня контейнера переопределяет аналогичный контекст уровня пода. Он описывается в разделе `containers` манифеста пода, а его параметры не могут быть применены к томам, которые остаются на уровне пода.

Вот как выглядит контекст безопасности уровня контейнера:

```

apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    - name: hello-world-container
      # Определение контейнера
      # ...
      securityContext:
        privileged: true
        seLinuxOptions:
          level: "s0:c123,c456"

```

Защита кластера с помощью AppArmor

AppArmor — это модуль безопасности в ядре Linux. С его помощью можно ограничить набор ресурсов, доступных процессу внутри контейнера, например доступ к сети и файлам, а также возможности Linux. Для конфигурации AppArmor используются профили.

Требования

Поддержка AppArmor появилась в Kubernetes 1.4 в статусе бета. Она доступна не во всех операционных системах, поэтому, чтобы воспользоваться преимуществами данного модуля, следует выбрать подходящий дистрибутив. Ubuntu и SUSE Linux поддерживают и используют AppArmor по умолчанию. В других дистрибутивах поддержка опциональна. Чтобы проверить, включен ли модуль AppArmor, введите следующую команду:

```
cat /sys/module/apparmor/parameters/enabled
Y
```

Если она вернет Y, модуль включен.

Профиль должен быть загружен в ядро. Проверьте файл `/sys/kernel/security/apparmor/profiles`.

Стоит также отметить, что сейчас AppArmor поддерживается лишь в среде выполнения Docker.

Защита пода с помощью AppArmor

AppArmor все еще находится на стадии бета, поэтому метаданные указываются в виде аннотаций, а не как поля `bonafide`. Это изменится, когда выйдет стабильная версия.

Чтобы применить профиль к контейнеру, добавьте следующую аннотацию:

```
container.apparmor.security.beta.kubernetes.io/<название-контейнера>:
<profileref>
```

В качестве базового можно указать либо профиль по умолчанию `runtime/default`, либо файл узла `localhost/<название-профиля>`.

Далее приведен пример профиля, который предотвращает запись в файлы:

```
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
    #include <abstractions/base>
    file,

    # Запрет на запись в любые файлы
    deny /** w,
}
```

Модуль AppArmor не является ресурсом Kubernetes, поэтому в нем не используются знакомые вам форматы файлов YAML и JSON.

Чтобы убедиться в корректном подключении профиля, проверьте атрибуты процесса 1:

```
kubect1 exec <название-пода> cat /proc/1/attr/current
```

По умолчанию поды можно разворачивать на любом узле кластера. Это означает, что профиль должен быть загружен на каждом узле. Таков классический сценарий для использования DaemonSet.

Ручное создание профилей AppArmor

Ручное создание профилей AppArmor — важный навык. Здесь вам могут пригодиться инструменты `aa-genprof` и `aa-logprof`, которые генерируют профиль и помогают его откорректировать, запуская ваше приложение с AppArmor в режиме с подробным выводом. Они отслеживают работу вашего приложения и предупреждения, выдаваемые AppArmor, одновременно создавая подходящий профиль. Этот с виду неуклюжий подход вполне действенный.

Но больше всего мне нравится утилита `bane` (<https://github.com/jessfraz/bane>), которая генерирует профили AppArmor из файла на языке, основанном на синтаксисе TOML. Описания `bane` выглядят довольно выразительными. Далее представлен небольшой отрывок:

```
Name = "nginx-sample"
[Filesystem]
# каталоги контейнера, доступные только для чтения
ReadOnlyPaths = [
    "/bin/**",
    "/boot/**",
    "/dev/**",
]
# каталоги, операции записи в которые вы хотите добавлять в журнал
LogOnWritePaths = [
    "/"
]
# разрешенные возможности
[Capabilities]
Allow = [
    "chown",
    "setuid",
]
[Network]
Raw = false
Packet = false
Protocols = [
    "tcp",
    "udp",
    "icmp"
]
```

Сгенерированный профиль получится малопонятным.

Политики безопасности для пода

В Kubernetes 1.4 появилась бета-версия ресурса *Pod Security Policy* (политика безопасности пода, PSP). Его нужно включить вручную вместе с контролем доступа PSP. Этот ресурс, определенный на уровне кластера, описывает контекст безопасности для подов. PSP имеет определенные особенности по сравнению с заданием защищенного содержимого вручную, как мы делали ранее.

- ❑ Он применяет одну и ту же политику к нескольким подам и контейнерам.
- ❑ Он позволяет администратору контролировать создание пода, не давая пользователям указывать некорректный контекст безопасности.
- ❑ Он динамически генерирует для пода разное защищенное содержимое с использованием контроллера доступа.

Политики PSP выводят концепцию контекста безопасности на новый уровень. Обычно количество подов (или, скорее, их шаблонов) значительно превышает количество политик безопасности. Это означает, что многие шаблоны подов и контейнеры будут иметь одну и ту же политику. Без PSP вам пришлось бы прописывать все это вручную для каждого манифеста пода.

Рассмотрим пример политики PSP, которая не накладывает никаких запретов:

```
{
  "kind": "PodSecurityPolicy",
  "apiVersion": "policy/v1beta1",
  "metadata": {
    "name": "permissive"
  },
  "spec": {
    "seLinux": {
      "rule": "RunAsAny"
    },
    "supplementalGroups": {
      "rule": "RunAsAny"
    },
    "runAsUser": {
      "rule": "RunAsAny"
    },
    "fsGroup": {
      "rule": "RunAsAny"
    },
    "volumes": ["*"]
  }
}
```

Авторизация политик безопасности для подов через RBAC

Это рекомендуемый способ включения политик безопасности. Создадим роль `clusterRole` (можно просто `Role`), чтобы разрешить использование определенных политик. Она должна выглядеть следующим образом:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: <role name>
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs: ['use']
  resourceNames:
  - <list of policies to authorize>
```

Теперь привяжем роль кластера к авторизованным пользователям:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: <binding name>
roleRef:
  kind: ClusterRole
  name: <role name>
  apiGroup: rbac.authorization.k8s.io
subjects:
# Авторизируем определенные служебные учетные записи:
- kind: ServiceAccount
  name: <authorized service account name>
  namespace: <authorized pod namespace>
# Авторизируем определенных пользователей (не рекомендуется):
- kind: User
  apiGroup: rbac.authorization.k8s.io
  name: <authorized user name>
```

Если роли используются лишь в качестве привязки, а не на уровне кластера, их действие ограничено подами в соответствующем пространстве имен. Их можно применять вместе с системными группами, чтобы открыть доступ для всех подов, запущенных в одном пространстве имен:

```
# Авторизируем все служебные учетные записи в пространстве имен:
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:serviceaccounts
# Или авторизируем всех аутентифицированных пользователей (что то же самое):
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:authenticated
```

Управление сетевыми политиками

Безопасность узлов, подов и контейнеров — первоочередная задача, но обеспечить только ее недостаточно. Для проектирования безопасных кластеров Kubernetes с поддержкой мультиарендности и минимальным влиянием потенциальных

уязвимостей крайне важна сегментация сети. Многоуровневая защита требует изолировать части системы, которые не общаются друг с другом, — это позволяет тщательно контролировать направление трафика, его протоколы и порты.

Сетевые политики обеспечивают тонкий контроль и надлежащую сегментацию сети вашего кластера. По своей сути сетевая политика — это набор правил для брандмауэра, которые применяются к пространствам имен и подам с определенными метками. Это очень гибкий подход, так как метки могут определять сегменты виртуальной сети, и управлять ими можно как обычными ресурсами Kubernetes.

Выбор совместимого сетевого решения

Некоторые серверы не поддерживают сетевые политики. Один из примеров — популярное решение Flannel.

Далее перечислены серверы, совместимые с сетевыми политиками:

- ❑ Calico;
- ❑ WeaveNet;
- ❑ Canal;
- ❑ Cillium;
- ❑ Kube-Router;
- ❑ Romana.

Определение сетевой политики

Сетевая политика определяется в стандартном манифесте формата YAML. Далее приведен пример:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: the-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: cool-project
      - podSelector:
          matchLabels:
            role: frontend
  ports:
    - protocol: tcp
      port: 6379
```


Раздел `spec` содержит два важных подраздела — `podSelector` и `ingress`. Первый определяет, к каким подам применяется данная сетевая политика, второй описывает другие пространства имен и поды, к которым могут обращаться эти экземпляры, а также протоколы и порты, используемые для этого.

В нашем примере селектор `pod` применяет сетевую политику ко всем подам с меткой `role: db`. Раздел `ingress` содержит подраздел `from` с селекторами `namespace` и `pod`. Все пространства имен кластера с меткой `project: cool-project`, а также все их поды, помеченные как `role: frontend`, могут обращаться к целевому экземпляру пода с меткой `role: db`. В разделе `ports` перечислены пары «протокол — порт», которые уточняют допустимые протоколы и порты. В данном случае разрешены протокол `tcp` и порт `6379` (стандартный для Redis).



Заметьте, что эта политика действует во всем кластере, поэтому доступ к целевому пространству имен будут иметь поды из разных пространств. Текущее пространство имен добавляется по умолчанию, поэтому, даже если оно не помечено как `project: cool-project`, поды с меткой `role: frontend` все равно получают доступ.

Важно понимать, что сетевая политика оперирует «белыми списками». Иначе говоря, изначально доступ закрыт полностью, а затем для подов с подходящими метками открываются отдельные протоколы и порты. Таким образом, если ваше сетевое решение не поддерживает сетевые политики, любой доступ будет запрещен.

Еще одно последствие такого подхода состоит в том, что при наличии нескольких сетевых политик применяется набор из всех их правил. Например, если в рамках одного пода две разные политики открывают порты `1234` и `5678`, это означает, что под может обращаться к любому из них.

Ограничение выхода во внешние сети

В Kubernetes 1.8 появилась поддержка сетевой политики `egress`, которая позволяет контролировать исходящий трафик. Приведенный далее пример закрывает доступ к внешнему IP-адресу `1.2.3.4`. Благодаря параметру `order: 999` данная политика применяется раньше других:

```
apiVersion: v1
kind: policy
metadata:
  name: default-deny-egress
spec:
  order: 999
  egress:
  - action: deny
    destination:
      net: 1.2.3.4
    source: {}
```

Политики, действующие в разных пространствах имен

Если кластер поделен на несколько пространств имен, вам может потребоваться взаимодействие подов из разных пространств. Чтобы разрешить доступ из нескольких пространств имен, укажите в описании своей сетевой политики поле `ingress.namespaceSelector`. Это может быть удобно, если, к примеру, у вас есть промышленное и тестовое пространства имен и вы хотите периодически переносить данные из одного в другое.

Использование объектов `secret`

Объекты `secret` являются важнейшим элементом защиты системы. Это могут быть учетные данные, такие как имя пользователя и пароль, токены доступа, API-ключи или ключи шифрования. Обычно эти объекты невелики. Если у вас много информации, которую нужно защитить, следует ее зашифровать и поместить в объект `secret` соответствующий ключ.

Хранение объектов `secret` в Kubernetes

Когда-то платформа Kubernetes хранила объекты `secret` в `etcd` в открытом виде. Из-за этого доступ к `etcd` был ограничен и тщательно защищен. В Kubernetes 1.7 появилась возможность шифрования объектов `secret` через REST (если они хранятся в `etcd`).

Управление объектами `secret` происходит на уровне пространства имен. К подам их можно подключать через файлы, секретные тома или в качестве переменных окружения. С точки зрения безопасности это означает, что любые пользователи или сервисы, которые могут создать под в определенном пространстве имен, имеют доступ ко всем объектам `secret` этого пространства. Если вы хотите сделать более строгим доступ к своему объекту, поместите последний в пространство имен с ограниченным набором пользователей или сервисов.

При подключении объекта `secret` к поду он помещается в `tmpfs` и никогда не записывается на диск. Для взаимодействия между `kubelet` и API-сервером обычно используется TLS, поэтому во время передачи объект `secret` находится в безопасности.

Настройка шифрования для REST

При запуске API-сервера следует указать такой параметр:

```
--experimental-encryption-provider-config <файл с настройками шифрования>
```

Далее приведен пример конфигурации шифрования:

```
kind: EncryptionConfig
apiVersion: v1
resources:
```

```

- resources:
  - secrets
  providers:
  - identity: {}
  - aesgcm:
    keys:
    - name: key1
      secret: c2VjcmV0IGlzIHNlY3VyZQ==
    - name: key2
      secret: dGhpcyBpcyBwYXNzd29yZA==
  - aescbc:
    keys:
    - name: key1
      secret: c2VjcmV0IGlzIHNlY3VyZQ==
    - name: key2
      secret: dGhpcyBpcyBwYXNzd29yZA==
  - secretbox:
    keys:
    - name: key1
      secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=

```

Создание объектов secret

Объекты `secret` должны быть готовы к моменту написания пода, который их использует, иначе этот процесс завершится неудачно.

Объект `secret` можно создать с помощью следующей команды:

```
kubectl create secret
```

Далее создается стандартный объект под названием `hush-hash`, который содержит два ключа — имя пользователя и пароль:

```
> kubectl create secret generic hush-hush --from-literal=username=tobias
--from-literal=password=cutoffs
```

Готовый объект имеет тип `Opaque`:

```
> kubectl describe secrets/hush-hush
Name:          hush-hush
Namespace:     default
Labels:        <none>
Annotations:   <none>
Type:          Opaque
Data
====
password:      7 bytes
username:      6 bytes
```

Вы можете указать параметр `--from-file` вместо `--from-literal`, чтобы создать объект `secret` из файла. Это можно выполнить и вручную, если закодировать содержимое объекта в формате `base64`.

К именам ключей внутри объектов применяются те же правила, что и к поддоменам в DNS (без начальной точки).

Декодирование объектов secret

Получить содержимое объекта `secret` можно с помощью команды `kubectl get secret:`

```
> kubectl get secrets/hush-hush -o yaml
apiVersion: v1
data:
  password: Y3V0b2Zmcw==
  username: dG9iaWFz
kind: Secret
metadata:
  creationTimestamp: 2018-01-15T23:43:50Z
  name: hush-hush
  namespace: default
  resourceVersion: "2030851"
  selfLink: /api/v1/namespaces/default/secrets/hush-hush
  uid: f04641ef-fa4d-11e7-beab-080027c94384
type: Opaque
The values are base64-encoded. You need to decode them yourself:
> echo "Y3V0b2Zmcw==" | base64 --decode
cutoofs
```

Значения хранятся в кодировке `base64`. Вам придется декодировать их самостоятельно:

```
> echo "Y3V0b2Zmcw==" | base64 --decode
cutoofs
```

Использование объектов secret в контейнерах

Контейнеры могут обращаться к объектам `secret` через файлы, подключая тома из подов. Для этого можно задействовать и переменные окружения. Еще один вариант — прямой доступ к API у Kubernetes (если служебная учетная запись контейнера имеет подходящие права) или применение команды `kubectl get secret`.

Чтобы задействовать объект `secret`, подключенный в виде тома, следует объявить этот том в манифесте пода и указать в спецификации контейнера:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "pod-with-secret",
    "namespace": "default"
  },
  "spec": {
```

```

"containers": [{
  "name": "the-container",
  "image": "redis",
  "volumeMounts": [{
    "name": "secret-volume",
    "mountPath": "/mnt/secret-volume",
    "readOnly": true
  }]
}],
"volumes": [{
  "name": "secret-volume",
  "secret": {
    "secretName": "hush-hush"
  }
}]
}
}
}

```

Название тома пода (`secret-volume`) привязывает его к точке подключения в контейнере. Разные контейнеры могут подключить один и тот же том.

После запуска этого пода имя пользователя и пароль будут доступны в виде файлов внутри каталога `/etc/secret-volume`:

```

> kubectl exec pod-with-secret cat /mnt/secret-volume/username
tobias
> kubectl exec pod-with-secret cat /mnt/secret-volume/password
cutoffs

```

Многопользовательские кластеры

В этом разделе мы затронем тему размещения на едином кластере систем, принадлежащих разным пользователям или сообществам. Смысл данного приема в том, что эти пользователи полностью изолированы друг от друга и могут даже не догадываться, что делят кластер с кем-то еще. У каждого сообщества пользователей будут свои ресурсы, и между ними не происходит никакого взаимодействия (разве что через публичные конечные точки). Отличное выражение этой идеи — концепция пространств имен в Kubernetes.

Необходимость в многопользовательских кластерах

Зачем размещать разных изолированных пользователей или развертывания на одном кластере? Разве не было бы проще выделить отдельные кластеры? Этому есть две основные причины: значительные издержки и сложность обслуживания. Если вы хотите развернуть множество относительно мелких систем на отдельных кластерах, во всех случаях вам придется выделять отдельный ведущий узел и, возможно, кластер etcd, состоящий из трех узлов. Это может оказаться накладно.

Сложность обслуживания тоже важна. Управление десятками, сотнями или тысячами кластеров — задача не из легких. Все обновления и заплатки необходимо применять к каждому кластеру. Что-то может пойти не так, и у вас на руках окажется целая армия кластеров, состояния которых могут немного различаться. Общие кластерные операции могут оказаться еще сложнее. Вам придется заниматься агрегацией и написанием собственных инструментов для выполнения определенных действий и сбора данных.

Рассмотрим некоторые сценарии использования множественных изолированных сообществ/систем и требования, которые к ним предъявляются.

- ☐ Платформа или провайдер для предоставления *<вписать нужное>* в качестве услуги.
- ☐ Управление отдельными тестовыми и промышленными средами.
- ☐ Делегирование ответственности администраторам сообществ/систем.
- ☐ Применение квот и лимитов на ресурсы для каждого сообщества.
- ☐ Пользователи могут видеть только ресурсы своего собственного сообщества.

Безопасная мультиарендность на основе пространств имен

Пространства имен в Kubernetes не случайно являются идеальным решением для безопасных мультиарендных кластеров — это было одной из причин их создания.

Вы можете легко добавлять собственные пространства имен к стандартному и встроенной системе kube. Далее показан файл YAML, который создает новое пространство имен под названием `custom-namespace`. Для этого достаточно метаданных с одним лишь полем `name`. Проще не придумаешь:

```
apiVersion: v1
kind: Namespace
metadata:
name: custom-namespace
```

Создадим это пространство имен:

```
> kubectl create -f custom-namespace.yaml
namespace "custom-namespace" created
> kubectl get namespaces
NAME                STATUS      AGE
custom-namespace    Active     39s
default              Active     32d
kube-system          Active     32d
```

Поле состояния может быть равно `active` или `terminating`. Состояние `terminating` наступает при удалении пространства имен, в этом случае вы не смо-

жете создавать в нем новые ресурсы. Это упрощает освобождение ресурсов пространства имен и гарантирует его полное удаление. В противном случае контроллер репликации мог бы создать новые поды вместо удаленных.

Для работы с пространством имен нужно в команде `kubectl` указать аргумент `--namespace`:

```
> kubectl create -f some-pod.yaml --namespace=custom-namespace
pod "some-pod" created
```

В новом пространстве имен значится лишь один под, который мы только что создали:

```
> kubectl get pods --namespace=custom-namespace
NAME          READY   STATUS    RESTARTS   AGE
some-pod      1/1     Running   0           6m
```

Если опустить аргумент `--namespace`, мы получим список подов в стандартном пространстве имен:

```
> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
echo-3580479493-n66n4              1/1     Running   16          32d
leader-elect-191609294-1t95t       1/1     Running   4           9d
leader-elect-191609294-m6fb6       1/1     Running   4           9d
leader-elect-191609294-piu8p       1/1     Running   4           9d
pod-with-secret                     1/1     Running   1           1h
```

Подводные камни работы с пространствами имен

Пространства имен — отличная вещь, но они могут добавить головной боли. Работая лишь со стандартным пространством имен, аргумент `--namespace` можно опустить. Но если пространств несколько, вы должны указывать его в каждой команде. Это может быть обременительным, но никакой опасности здесь нет. Однако если некоторые пользователи, например администраторы кластера, имеют доступ к разным пространствам имен, они могут случайно отредактировать данные или выполнить запрос не в том пространстве. Чтобы избежать таких ситуаций, лучше всего полностью изолировать разные пространства имен и предусмотреть для доступа к ним отдельные учетные записи и полномочия.

Кроме того, некоторые инструменты могут помочь прояснить, с каким пространством имен вы сейчас работаете, например выводя его в приглашении командной строки или на видном месте в веб-интерфейсе.

Убедитесь в том, что пользователи, которые могут работать с отдельным пространством имен, не имеют доступа к стандартному пространству. В противном случае, если они забудут указать аргумент `--namespace`, их действия будут незаметно применяться к пространству имен по умолчанию.

Резюме

В этой главе мы поговорили о различных проблемах с безопасностью, с которыми сталкиваются разработчики и администраторы при построении систем и развертывании приложений в кластерах Kubernetes. А также рассмотрели множество возможностей и гибких подключаемых моделей безопасности, обеспечиваемых разными средствами для ограничения и контроля за контейнерами, подами и узлами. Kubernetes предлагает разносторонние решения большинства проблем, связанных с безопасностью. В этой области можно ожидать новых улучшений, так как подсистемы вроде AppArmor и различные дополнения выходят из состояния альфа/бета и становятся общедоступными. В конце вы узнали, как с помощью пространств имен можно обеспечить поддержку нескольких сообществ пользователей и систем в одном и том же кластере Kubernetes.

В следующей главе мы рассмотрим множество ресурсов и концепций из мира Kubernetes, будет показано, как они могут эффективно использоваться вместе и по отдельности. Объектная модель Kubernetes базируется на прочном фундаменте, состоящем из небольшого количества концепций, таких как ресурсы, манифесты и метаданные. Это позволяет добиться расширяемости и одновременно необыкновенной согласованности, благодаря чему разработчики и администраторы получают набор разнообразных возможностей.

6

Использование критически важных ресурсов Kubernetes

В этой главе мы спроектируем огромную систему, которая по максимуму будет задействовать возможности платформы Kubernetes и ее способность к масштабированию. Система Hue предназначена для создания всезнающего и всемогущего цифрового помощника. Это будет ваша правая рука в цифровом мире. Очевидно, ей нужно будет хранить множество информации, взаимодействовать с внешними сервисами, реагировать на уведомления и события и быть достаточно умной, общаясь с вами.

Мы воспользуемся этой возможностью, чтобы поближе познакомиться с `kubectl` и другими инструментами. Кроме того, подробно разберем ресурсы, с которыми сталкивались ранее (поды) и которые еще не затрагивали (задания). По прочтении этой главы у вас сложится четкое представление о впечатляющих возможностях Kubernetes и о том, как на основе данной платформы создавать сложные системы.

Проектирование системы Hue

В этом разделе мы подготовим почву для дальнейшей разработки и определим область применения нашей удивительной системы Hue. Hue — не «большой брат», а, скорее, «братишка», который будет делать то, что вы ему позволите. Потенциал данной системы очень велик, но, чтобы не вызывать лишних опасений, вы сами сможете определить, насколько полезной она должна быть. Пристегните ремни!

Область применения Hue

Система Hue будет управлять вашим цифровым «я». Она будет знать о вас больше, чем вы сами. Далее перечислены некоторые сферы, в которых Hue вам способен помочь.

- ☐ Поиск и агрегация содержимого.
- ☐ Здоровье.

- ❑ Умный дом.
- ❑ Финансы, накопления, выход на пенсию, инвестиции.
- ❑ Офис.
- ❑ Социальная жизнь.
- ❑ Путешествия.
- ❑ Семья.
- ❑ **Умные напоминания и уведомления.** Подумаем о потенциальных возможностях. Ние будет знать не только о вас, но и о ваших друзьях, а также в общем о пользователях в других областях. Эта система будет обновлять свои модели в режиме реального времени — ее нельзя будет запутать устаревшими данными. Она будет действовать от вашего имени, предоставляя подходящую информацию и непрерывно изучая ваши предпочтения. Сможет посоветовать новые сериалы или книги, которые вам понравятся, заказать столик в ресторане с учетом вашего расписания и доступности семьи или друзей, а также управлять техникой в вашем доме.
- ❑ **Безопасность, проверка подлинности и конфиденциальность.** Ние — ваш представитель в Интернете. Последствия от похищения учетных данных этой системы или всего лишь перехват вашего взаимодействия с ней будут губительными, что может разрушить доверие потенциальных пользователей к Ние. Создадим такую систему, которую реально обуздать в любой момент. Вот несколько идей в этой области.
 - Надежная проверка подлинности с помощью многофакторной авторизации через отдельное устройство, включая множественные биометрические проверки.
 - Частая замена учетных данных.
 - Быстрая служебная остановка системы и проверка подлинности всех внешних сервисов (потребует наличия оригинальных доказательств подлинности для каждого провайдера).
 - Взаимодействие сервера Ние с внешними сервисами с помощью краткосрочных токенов.
 - Проектирование Ние в виде набора слабо связанных микросервисов.

Архитектура Ние должна быть гибкой и поддерживать огромное разнообразие функций. Она также должна быть расширяемой, с постоянным обновлением существующих возможностей и внешних сервисов и интеграцией новых. В таком масштабном проекте, где все возможности и сервисы абсолютно независимы друг от друга, а единственным средством взаимодействия между ними являются четко определенные API (стандартные или доступные для обнаружения), напрашивается использование микросервисов.

Компоненты Hue

Перед началом путешествия в мир микросервисов рассмотрим типы компонентов, которые понадобятся для построения Hue.

- ❑ **Профиль пользователя.** Это основной компонент с множеством подкомпонентов. Он олицетворяет собой пользователей, их предпочтения, историю активности в рамках системы и все, что Hue о них знает.
- ❑ **Социальный граф.** Этот компонент моделирует сеть взаимодействий между пользователями в разных областях. Каждый пользователь участвует в нескольких сетях: социальных (таких как Facebook и Twitter), профессиональных, сетях, посвященных определенным хобби, и добровольных сообществах. Некоторые из этих сетей узкоспециализированные, и Hue способен их структурировать для удобства пользователей. Благодаря многофункциональным профилям можно будет улучшить взаимодействие между пользователями, не выставляя напоказ их личные данные.
- ❑ **Учетные записи.** Как уже упоминалось, управление учетными записями имеет большое значение и заслуживает реализации в виде отдельного компонента. У пользователя может быть несколько профилей, принадлежащих разным виртуальным личностям. Это связано, например, с тем, что пользователи предпочитают не смешивать медицинские и социальные записи, чтобы друзья не могли получить доступ к частной информации об их здоровье.
- ❑ **Авторизация.** Это крайне важный компонент, который дает Hue явное разрешение на выполнение определенных действий и сбор различных данных от своего имени. Сюда входит доступ к физическим устройствам, внешним учетным записям и уровням инициативы.
- ❑ **Внешние сервисы.** Hue — агрегатор внешних сервисов. Эта система не пытается заменить собой ваш банк, поликлинику или социальные сети. Она собирает множество метаданных о ваших занятиях, но сама информация будет оставаться во внешних сервисах. Каждый внешний сервис потребует реализации отдельного компонента для работы с его API и политиками. Если API отсутствует, Hue эмулирует пользователя путем автоматизации веб-страниц и родных приложений.
- ❑ **Центральный хаб.** Одно из самых полезных свойств системы Hue — ее умение действовать от имени пользователя. Чтобы делать это эффективно, она должна знать о различных событиях. Например, если система запланировала для вас отпуск, но затем ей стало известно о более дешевых билетах, она способна автоматически откорректировать маршрут или спросить вашего разрешения на это действие. Существует бесконечное множество подобной информации, и для ее сбора нужен центральный хаб. Его можно будет расширять, но другие части Hue смогут обращаться к нему через стандартный интерфейс.

- ❑ **Универсальный контроллер.** Данный компонент работает рука об руку с центральным хабом. Ние нужно выполнять от вашего имени разные действия, такие как бронирование билетов. Для этого необходим универсальный контроллер, который можно расширять для поддержки определенных функций. При этом другие компоненты, такие как диспетчер учетных записей и авторизатор, смогут обращаться к нему через унифицированный интерфейс.
- ❑ **Самообучающийся агент.** Это мозг Ние. Он будет постоянно отслеживать все ваши взаимодействия (с вашего разрешения) и адаптировать под вас свою модель. Это позволит системе Ние развиваться и становиться более полезной: предугадывать ваши потребности и интересы, предоставлять лучший выбор, выдавать более уместную информацию в подходящий момент, не раздражать вас по пустякам и не брать на себя слишком много.

Микросервисы в Ние

Все компоненты очень сложны. Некоторые из них, например внешние сервисы, центральный хаб и универсальный контроллер, будут работать с сотнями или даже тысячами других внешних сервисов, постоянные изменения которых Ние не контролирует. Даже самообучающемуся агенту нужно собирать сведения о предпочтениях пользователя в разных областях. Подходящим решением здесь могут стать микросервисы, которые позволят системе Ние постепенно эволюционировать и наращивать изолированные возможности, не становясь при этом слишком сложной. Каждый микросервис взаимодействует с универсальными инфраструктурными сервисами Ние через стандартные интерфейсы и при необходимости — с другими элементами системы, используя четко определенные API с поддержкой версий. Внешне все микросервисы остаются управляемыми, а взаимодействие между ними основано на стандартных устоявшихся методиках.

- ❑ **Дополнения.** Являются ключом к расширяемости Ние и позволяют не плодить лишние интерфейсы. Часто их приходится объединять в цепочки, пересекающие разные уровни абстракции. Например, при добавлении в Ние интеграции с YouTube можно собрать множество дополнительной информации: ваши каналы, любимые ролики, рекомендации и просмотренные видео. Чтобы отобразить эти данные и позволить пользователям работать с ними, понадобятся дополнения для разных компонентов и в конечном счете — для пользовательского интерфейса. Продуманная архитектура позволит агрегировать категории действий, такие как рекомендации, выборки и отложенные уведомления, во множество разных сервисов.

У дополнений есть одна замечательная особенность: их может разрабатывать кто угодно. Вначале написанием дополнений будет заниматься команда разработчиков Ние, но с ростом популярности внешние сервисы сами захотят интегрироваться с Ние и начнут создавать собственные дополнения.

Это неминуемо приведет к появлению целой экосистемы с регистрацией, одобрением и курированием дополнений.

❑ **Хранилища данных.** Для управления информацией и метаданными Hue понадобятся по несколько серверов в каждой из перечисленных далее категорий:

- реляционные базы данных;
- графовые базы данных;
- базы данных на основе временных рядов;
- кэширование в памяти.

Ввиду огромного размера Hue каждая из этих баз данных должна быть кластеризованной и распределенной.

❑ **Микросервисы без хранения состояния.** Микросервисы по большей части не должны хранить свое состояние. Это позволяет им быстро запускаться и останавливаться, а также в случае необходимости мигрировать в другую инфраструктуру. Состояние будет храниться в базе данных, а доступ к нему можно получить с помощью краткосрочных токенов.

❑ **Взаимодействие на основе очереди.** Все микросервисы должны общаться друг с другом. Пользователи будут просить Hue выполнить какие-то действия от их имени. Внешние сервисы станут уведомлять Hue о разных событиях. Идеальное решение — сочетание очередей с микросервисами, не хранящими свое состояние. Множество экземпляров разных микросервисов подписываются на подходящие события или запросы, предоставляемые разными очередями, и отвечают на них по мере поступления. Это здравый подход, работу легко масштабировать. Каждый компонент может быть избыточным и высокодоступным, и даже если он выйдет из строя, система в целом останется крайне устойчивой к отказам.

Очередь можно использовать как для вызова удаленных асинхронных процедур (RPC), так и для взаимодействия в стиле «запрос — ответ». В последнем случае вызывающий экземпляр предоставляет имя частной очереди, а отвечающая сторона помещает в нее свой ответ.

Планирование рабочих процессов

В Hue должна реализовываться поддержка рабочих процессов. Типичный рабочий процесс получает высокоуровневое задание (например, записаться на прием к стоматологу), извлекает сведения о зубном враче пользователя и его рабочем графике, сопоставляет их с расписанием самого пользователя, выбирает один из нескольких вариантов, опционально запрашивает подтверждение, записывается на прием и подготавливает напоминание. Рабочие процессы могут быть полностью автоматическими или требовать участия человека. Или они связаны с тратой денег.

Автоматические рабочие процессы

Автоматические рабочие процессы не требуют человеческого вмешательства. Hue обладает всеми полномочиями, необходимыми для выполнения задачи от начала до конца. Чем большую автономность от пользователя получит система, тем более эффективной она станет. У пользователей должна быть возможность просматривать и проверять все рабочие процессы — как прошлые, так и текущие.

Рабочие процессы с участием человека

Этот вид рабочих процессов подразумевает взаимодействие с человеком. В большинстве случаев пользователю нужно лишь выбрать из нескольких вариантов или одобрить какое-то действие, но нельзя исключать вовлечение живых людей на стороне внешнего сервиса. Например, чтобы записаться на прием к зубному врачу, вам, возможно, придется согласовать подходящее время с его секретарем.

Рабочие процессы финансового характера

Некоторые рабочие процессы, такие как коммунальные платежи или покупка подарка, требуют денежных расходов. Теоретически Hue может получить неограниченный доступ к банковскому счету пользователя, однако для большинства, скорее всего, предпочтительной будет возможность ограничивать бюджет для разных процессов или вручную подтверждать трату денег.

Использование Kubernetes для построения системы Hue

В этом разделе вы познакомитесь с различными ресурсами Kubernetes и узнаете, какую роль они могут сыграть при создании Hue. Вначале мы подробнее рассмотрим гибкую утилиту `kubectl`, а затем перейдем к таким темам, как: долгоиграющие процессы в Kubernetes, открытие внутреннего и внешнего доступа к сервисам, использование пространств имен для ограничения доступа, запуск узкоспециализированных заданий и интеграция компонентов, не принадлежащих кластеру. Hue — большой проект, поэтому вместо построения полноценной платформы мы рассмотрим эти идеи на примере локального кластера Minikube.

Эффективное применение `kubectl`

Утилита `Kubectl` способна делать с кластером все, что угодно. Для этого она подключается к API у кластера. `Kubectl` считывает ваш файл `.kube/config`, который

содержит информацию, необходимую для подключения к одному или нескольким кластерам. Все ее команды можно разделить на несколько категорий.

- ❑ **Общие команды.** Манипулируют ресурсами общего характера: `create`, `get`, `delete`, `run`, `apply`, `patch`, `replace` и т. д.
- ❑ **Команды для управления кластером.** Выполняют высокоуровневые манипуляции с узлами и кластером: `cluster-info`, `certificate`, `drain` и т. д.
- ❑ **Отладочные команды.** `describe`, `logs`, `attach`, `exec` и т. д.
- ❑ **Команды развертывания.** Отвечают за развертывание и масштабирование: `rollout`, `scale`, `auto-scale` и т. д.
- ❑ **Команды для работы с параметрами.** Работают с метками и аннотациями: `label`, `annotate` и т. д.
- ❑ **Прочие команды.** `help`, `config` и `version`.

Просмотреть конфигурацию Kubernetes можно с помощью команды `config view`. Далее представлена конфигурация кластера Minikube:

```
~/minikube > k config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /Users/gigi.sayfan/.minikube/ca.crt
    server: https://192.168.99.100:8443
    name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
    name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /Users/gigi.sayfan/.minikube/client.crt
    client-key: /Users/gigi.sayfan/.minikube/client.key
```

Файлы конфигурации ресурсов в kubectl

Многие операции в `kubectl`, такие как `create`, требуют сложного иерархического вывода (на самом деле это требование API). `Kubectl` использует конфигурационные файлы форматов `YAML` и `JSON`. Далее показана конфигурация в формате `YAML` для создания пода:

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: ""
  labels:
    name: ""
  namespace: ""
  annotations: []
  generateName: ""
spec:
  ...

```

- ❑ **apiVersion.** Важнейшие API Kubernetes продолжают развиваться и могут поддерживать разные версии одного и того же ресурса в разных версиях API.
- ❑ **kind.** Благодаря этому обязательному полю платформа Kubernetes знает, с ресурсом какого типа имеет дело. В данном случае это под.
- ❑ **metadata.** Здесь содержится множество информации о поде и контексте, в котором он работает:
 - **name** — уникальный идентификатор пода в рамках пространства имен;
 - **labels** — можно указать несколько меток;
 - **namespace** — пространство имен, которому принадлежит под;
 - **annotations** — список аннотаций, доступных при запросе.
- ❑ **spec.** Это шаблон, который содержит всю информацию, необходимую для запуска пода. Он может быть довольно сложным, поэтому разобьем его на несколько частей:

```

"spec": {
  "containers": [
  ],
  "restartPolicy": "",
  "volumes": [
  ]
}

```

- ❑ **Container spec.** В этом поле перечислены спецификации контейнеров. Каждая спецификация имеет следующую структуру:

```

{
  "name": "",
  "image": "",
  "command": [
    ""
  ],
  "args": [
    ""
  ],
  "env": [
    {
      "name": "",
      "value": ""
    }
  ]
}

```



```
    }  
  ],  
  "imagePullPolicy": "",  
  "ports": [  
    {  
      "containerPort": 0,  
      "name": "",  
      "protocol": ""  
    }  
  ],  
  "resources": {  
    "cpu": ""  
    "memory": ""  
  }  
}
```

У каждого контейнера есть команда, которая, будучи указанной, заменяет собой команду Docker-образа. Имеются также аргументы и переменные окружения. И конечно, политика загрузки образа, порты и лимиты на ресурсы. Мы рассмотрели эти поля в предыдущих главах.

Развертывание долгоиграющих микросервисов в подах

Долгоиграющие микросервисы выполняются в подах и не должны хранить свое состояние. Посмотрим, как создаются поды для одного из микросервисов Hue. Позже мы повысим уровень абстракции и воспользуемся развертыванием.

Создание подов

Для начала разработаем внутренний сервис самообучающегося агента, используя обычный конфигурационный файл пода. Этот сервис не предназначен для доступа извне — он будет следить за очередью уведомлений и помещать полученные сведения в постоянное хранилище.

Нам нужен простой контейнер, который будет работать внутри пода. Далее представлен максимально простой Docker-файл, который будет имитировать самообучающийся сервис Hue:

```
FROM busybox  
CMD ash -c "echo 'Started...'; while true ; do sleep 10 ; done"
```

Он использует базовый образ `busybox`, возвращает в стандартный вывод строку `Started...` и входит в бесконечный цикл, который, как ни крути, долгоиграющий.

Я создал два образа Docker, `g1g1/hue-learn:v3.0` и `g1g1/huelearn:v4.0`, и поместил их в реестр Docker Hub (`g1g1` — мое имя пользователя):

```
docker build . -t g1g1/hue-learn:v3.0
docker build . -t g1g1/hue-learn:v4.0
docker push g1g1/hue-learn:v3.0
docker push g1g1/hue-learn:v4.0
```

Теперь эти образы можно загрузить в контейнеры внутри подов Hue.

Будем применять здесь формат YAML, так как его проще воспринимать. Далее представлен шаблон с метками `metadata`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
  labels:
    app: hue
    runtime-environment: production
    tier: internal-service
  annotations:
    version: "3.0"
```

Метки предназначены для идентификации наборов подов в рамках развертывания, поэтому для описания версии я использую аннотацию. Кроме того, метки нельзя изменять.

Дальше идет важное поле `containers`, которое определяет обязательные атрибуты `name` и `image` для каждого контейнера:

```
spec:
  containers:
  - name: hue-learner
    image: g1g1/hue-learn:v3.0
```

Раздел `resources` описывает ресурсы, которые требуются контейнеру, это позволяет планировать и выделять их более эффективным и компактным образом. Наш контейнер запрашивает 200 миллипроцессорных единиц (0,2 ядра) и 256 Мбайт памяти:

```
resources:
  requests:
    cpu: 200m
    memory: 256Mi
```

Раздел `environment` позволяет администратору кластера предоставить переменные окружения, которые будут доступны контейнеру. Здесь мы указываем, что обнаружение очереди и хранилища следует выполнять через DNS. В тестовой среде может использоваться другой метод обнаружения:

```
env:
- name: DISCOVER_QUEUE
  value: dns
- name: DISCOVER_STORE
  value: dns
```

Маркирование подов с помощью меток

Разумная маркировка подов — залог гибкого администрирования. Это позволяет разбивать кластер на лету, объединять микросервисы в группы, которые могут действовать в едином ключе, и опускаться на более низкие уровни, чтобы работать с разными их подмножествами.

Наш самообучающийся агент Hue, к примеру, имеет следующие метки:

- ❑ `runtime-environment` — производство (production).
- ❑ `tier` — внутренний сервис (internal-service).

С помощью поля `version` можно поддерживать сразу несколько версий. Если версии 2 и 3 должны выполняться одновременно (либо для обратной совместимости, либо на время миграции с v2 на v3), аннотации или метки позволяют независимо масштабировать поды и открывать доступ к разным их версиям. С помощью метки `runtime-environment` можно выполнять глобальные операции со всеми подами внутри определенной среды. Метку `tier` используют для объединения в очередь всех экземпляров подов, находящихся на одном уровне. Но это всего лишь примеры, дальше все зависит от вашего воображения.

Развертывание долгоиграющих процессов с помощью объектов `deployment`

В крупномасштабных системах поды всегда должны быть под контролем. Если по какой-то причине под перестает работать, его следует заменить, чтобы сохранить общую производительность. Вы можете сами создать контроллеры репликации и наборы реплик, но это чревато ошибками и вероятностью частичного отказа. Куда более логичным шагом было бы указывать количество необходимых реплик при запуске своих подов.

Развернем три экземпляра микросервиса с самообучающимся агентом, используя ресурс `deployment`. Объекты этого типа в Kubernetes 1.9 были объявлены стабильными:

```
apiVersion: apps/v1 (use apps/v1beta2 before 1.9)
kind: Deployment
metadata:
  name: hue-learn
  labels:
    app: hue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hue
  template:
    metadata:
      labels:
        app: hue
    spec: <same spec as in the pod template>
```

Поле `pod spec` идентично аналогичному разделу в конфигурационном файле пода, который мы использовали ранее.

Теперь выполним развертывание и проверим его состояние:

```
> kubectl create -f .\deployment.yaml
deployment "hue-learn" created
> kubectl get deployment hue-learn
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
hue-learn	3	3	3	3	4m

```
> kubectl get pods | grep hue-learn
```

NAME	READY	STATUS	RESTARTS	AGE
hue-learn-237202748-d770r	1/1	Running	0	2m
hue-learn-237202748-fwv2t	1/1	Running	0	2m
hue-learn-237202748-tpr4s	1/1	Running	0	2m

Намного больше информации о развертывании можно получить с помощью команды `kubectl describe`.

Обновление развертывания

Hue представляет собой крупную, непрерывно развивающуюся систему. Вам нужно постоянно выпускать новые версии. Одним из наименее проблемных способов являются плавающие обновления. Они полностью контролируются платформой Kubernetes, и чтобы их инициировать, достаточно изменить шаблон пода.

У нас все запущенные поды имеют версию 3.0:

```
> kubectl get pods -o json | jq .items[0].spec.containers[0].image
"3.0"
```

Обновим развертывание до версии 4.0. Отредактируем поле `version` в файле развертывания. Чтобы не спровоцировать ошибку, оставьте метки без изменений. Обычно меняются образ и связанные с ним метаданные в аннотациях. Для переключения на новую версию можно задействовать команду `apply`:

```
> kubectl apply -f hue-learn-deployment.yaml
deployment "hue-learn" updated
> kubectl get pods -o json | jq .items[0].spec.containers[0].image
"4.0"
```

Разделение внутренних и внешних сервисов

Внутренними называются сервисы, к которым напрямую обращаются лишь другие сервисы и задания в том же кластере или администраторы, входящие в систему и запускающие узкоспециализированные утилиты. Иногда внутренние сервисы вообще ни с кем не взаимодействуют, в этом случае они просто выполняют свою функцию и помещают результаты в постоянное хранилище, откуда их могут достать другие, не связанные с ними сервисы.

Но некоторые сервисы должны быть открыты для доступа со стороны пользователей или внешних программ. Рассмотрим фиктивный сервис Hue, который управляет списком напоминаний пользователя. Как и в случае с `hue-learn`, пример образа находится в Docker Hub под именем `hue-reminders`:

```
docker push g1g1/hue-reminders:v2.2
```

Развертывание внутреннего сервиса

Представленный далее объект `deployment` очень похож на тот, который использовался в самообучающемся агенте, я лишь удалил разделы `annotations`, `env` и `resources`, оставил только одну метку для экономии места и добавил в контейнер поле `ports`. Это важно, поскольку сервис должен предоставлять порт, через который к нему смогут обращаться другие сервисы:

```
apiVersion: apps/v1a1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
template:
  metadata:
    name: hue-reminders
    labels:
      app: hue-reminders
  spec:
    containers:
      - name: hue-reminders
        image: g1g1/hue-reminders:v2.2
        ports:
          - containerPort: 80
```

В результате развертывания в кластер будут добавлены два пода `reminders`:

```
> kubectl create -f hue-reminders-deployment.yaml
```

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hue-learn-56886758d8-h7vm7	1/1	Running	0	49m
hue-learn-56886758d8-lqptj	1/1	Running	0	49m
hue-learn-56886758d8-zwkqt	1/1	Running	0	49m
hue-reminders-75c88cdfcf-5xqtp	1/1	Running	0	50s
hue-reminders-75c88cdfcf-r6jsx	1/1	Running	0	50s

Итак, все поды запущены. Теоретически их могут найти и использовать напрямую другие сервисы, опционально сконфигурированные с применением внутренних IP-адресов, поскольку все они находятся в одном сетевом пространстве. Но такой подход не масштабируется. Все сервисы, которые обращаются к подам `reminders`, должны как-то узнавать об их замене и добавлении новых

экземпляров. Для того чтобы решить эту проблему, сервисы предоставляют единую точку доступа ко всем подам. Описание сервиса выглядит так:

```
apiVersion: v1
kind: Service
metadata:
  name: hue-reminders
  labels:
    app: hue-reminders
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    app: hue-reminders
```

У сервиса есть селектор, который выбирает все поды с соответствующими метками. Он также предоставляет порт, через который к нему могут обращаться другие сервисы (может не совпадать с портом контейнера).

Создание сервиса hue-reminders

Создадим сервис и вкратце его рассмотрим:

```
> kubectl create -f hue-reminders-service.yaml
service "hue-reminders" created
> kubectl describe svc hue-reminders
Name:                hue-reminders
Namespace:           default
Labels:              app=hue-reminders
Annotations:         <none>
Selector:            app=hue-reminders
Type:                ClusterIP
IP:                  10.108.163.209
Port:                <unset> 80/TCP
TargetPort:          80/TCP
Endpoints:           172.17.0.4:80,172.17.0.6:80
Session Affinity:    None
Events:              <none>
```

Сервис запустился и работает. Другие поды могут найти его через переменные окружения или DNS. Переменные окружения для всех сервисов устанавливаются в момент создания пода. Иначе говоря, если при создании сервиса ваш под уже запущен, придется его удалить и позволить Kubernetes воссоздать его, но уже с обновленными переменными (вы ведь создаете свои поды путем развертывания, не так ли?):

```
> kubectl exec hue-learn-56886758d8-fjzdd -- printenv | grep
HUE_REMINDERS_SERVICE
HUE_REMINDERS_SERVICE_PORT=80
HUE_REMINDERS_SERVICE_HOST=10.108.163.209
```

Намного проще использовать DNS-имя вашего сервиса:

```
<service name>.<namespace>.svc.cluster.local
> kubectl exec hue-learn-56886758d8-fjzdd -- nslookup hue-reminders
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local
Name:        hue-reminders
Address 1: 10.108.163.209 hue-reminders.default.svc.cluster.local
```

Выведение сервиса наружу

Сервис доступен внутри кластера. Для вывода его наружу Kubernetes предлагает два варианта.

- ☐ Сконфигурировать `NodePort` для прямого доступа.
- ☐ Сконфигурировать облачный балансировщик нагрузки, если он работает в облачной среде.

Прежде чем настраивать для сервиса возможность доступа извне, вы должны убедиться в его безопасности. В документации Kubernetes есть хороший пример, который показывает мельчайшие подробности: github.com/kubernetes/examples/blob/master/staging/https-nginx/README.md.

Основные принципы уже были рассмотрены в главе 5.

Далее показан раздел `спес` сервиса `hue-reminders` с доступом извне через `NodePort`:

```
спес:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
    selector:
      app: hue-reminders
```

Ingress. Ingress — это объект конфигурации Kubernetes, который позволяет вывести сервис наружу и заботиться о множестве мелких деталей. Он обладает следующими возможностями.

- ☐ Предоставляет сервису URL-адрес, который виден извне.
- ☐ Балансирует трафик.
- ☐ Разрывает SSL-соединение.
- ☐ Предоставляет виртуальный хостинг на основе DNS.

Для использования этих объектов нужен контроллер Ingress, запущенный в вашем кластере. Следует отметить, что эта подсистема все еще находится на стадии бета-тестирования и имеет множество ограничений. Если ваш кластер работает в GKE, скорее всего, никаких проблем не возникнет. В противном случае нужно проявить осторожность. Одно из нынешних ограничений контроллера Ingress — отсутствие поддержки масштабирования. Таким образом, он пока что не совсем подходит для системы Hue. Более подробно об Ingress поговорим в главе 10.

Вот как выглядит ресурс Ingress:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: fooSvc
          servicePort: 80
  - host: bar.baz.com
    http:
      paths:
      - path: /bar
        backend:
          serviceName: barSvc
          servicePort: 80
```

Контроллер Ingress интерпретирует этот запрос и создаст соответствующий конфигурационный файл для веб-сервера NGINX:

```
http {
  server {
    listen 80;
    server_name foo.bar.com;

    location /foo {
      proxy_pass http://fooSvc;
    }
  }
  server {
    listen 80;
    server_name bar.baz.com;
    location /bar {
      proxy_pass http://barSvc;
    }
  }
}
```

Можно создавать и другие контроллеры.

Ограничение доступа с помощью пространства имен

Проект Hue развивается довольно быстро. У нас есть несколько сотен микросервисов и примерно 100 разработчиков и инженеров DevOps, которые ими занимаются. Связанные между собой микросервисы объединяются в группы, многие из них автономные — они не имеют ни малейшего представления о других группах. Кроме того, мы имеем дело с некоторыми деликатными областями, такими как здоровье и финансы: желательно, чтобы доступ к ним контролировался более эффективно. Таким образом, мы подходим к теме пространств имен.

Создадим сервис hue-finance и поместим его в пространство имен под названием `restricted`.

Вот как выглядит YAML-файл нового пространства:

```
kind: Namespace
apiVersion: v1
metadata:
  name: restricted
  labels:
    name: restricted
> kubectl create -f restricted-namespace.yaml
namespace "restricted" created
```

После создания пространства имен мы должны сконфигурировать его контекст. Это позволит наложить ограничения доступа, которые будут касаться лишь данного пространства:

```
> kubectl config set-context restricted --namespace=restricted --
cluster=minikube --user=minikube
Context "restricted" set.
> kubectl config use-context restricted
Switched to context "restricted".
```

Проверим конфигурацию кластера:

```
> kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /Users/gigi.sayfan/.minikube/ca.crt
    server: https://192.168.99.100:8443
    name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
    name: minikube
- context:
    cluster: minikube
    namespace: restricted
    user: minikube
```

```

  name: restricted
current-context: restricted
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /Users/gigi.sayfan/.minikube/client.crt
    client-key: /Users/gigi.sayfan/.minikube/client.key

```

Как видите, в качестве текущего контекста указан `restricted`.

Теперь в пустом пространстве имен можно создать сервис `hue-finance`, который окажется там в полной изоляции:

```

> kubectl create -f hue-finance-deployment.yaml
deployment "hue-finance" created
> kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
hue-finance-7d4b84cc8d-gcjnz	1/1	Running	0	6s
hue-finance-7d4b84cc8d-tqvr9	1/1	Running	0	6s
hue-finance-7d4b84cc8d-zthdr	1/1	Running	0	6s

И не нужно переключать контексты. Также можно использовать параметры `-namespace=<namespace>` и `-all-namespaces`.

Запуск заданий

Система Hue состоит из множества «долгоиграющих» процессов, развернутых в виде микросервисов, есть у нее и много заданий, которые запускаются, достигают какой-то цели и завершаются. В Kubernetes для этого предусмотрен ресурс `job`, который управляет одним или несколькими подами и следит за тем, чтобы они работали до успешного выполнения поставленной задачи.

Далее показано задание, которое запускает процесс Python для вычисления факториала 5 (подсказка: он равен 120):

```

apiVersion: batch/v1
kind: Job
metadata:
  name: factorial5
spec:
  template:
    metadata:
      name: factorial5
    spec:
      containers:
      - name: factorial5
        image: python:3.6
        command: ["python",
                  "-c",
                  "import math; print(math.factorial(5)) "]
      restartPolicy: Never

```

Заметьте, что поле `restartPolicy` должно быть равно `Never` или `OnFailure`. Значение `Always`, установленное по умолчанию, в данном случае некорректно, поскольку задание не должно перезапускаться после успешного завершения.

Запустим это задание и проверим его состояние:

```
> kubectl create -f .\job.yaml
job "factorial5" created
> kubectl get jobs
NAME           DESIRED    SUCCESSFUL    AGE
factorial5     1          1             25s
```

Под завершеного задания по умолчанию не отображается. Вы должны использовать параметр `-show-all`:

```
> kubectl get pods -show-all
NAME                                READY   STATUS    RESTARTS   AGE
factorial5-ntp22                    0/1     Completed 0           2m
hue-finance-7d4b84cc8d-gcijnz      1/1     Running   0           9m
hue-finance-7d4b84cc8d-tqvr9       1/1     Running   0           8m
hue-finance-7d4b84cc8d-zthdr       1/1     Running   0           9m
```

Под с названием `factorial5` имеет статус `Completed` (Завершен). Проверим его вывод:

```
> kubectl logs factorial5-ntp22
120
```

Параллельное выполнение заданий

Задания можно запускать одновременно. В спецификации задания есть два поля, `completions` и `parallelism`. Первое по умолчанию равно 1. Если нужны несколько успешных завершений, это значение увеличивают. Поле `parallelism` определяет количество запускаемых подов. Задание запустит ровно столько подов, сколько требуется для его успешного завершения, даже если значение `parallelism` превышает значение `completions`.

Запустим еще одно задание, которое останавливается на 20 с, а потом успешно завершается три раза. Для этого будут запущены три пода, хотя в поле `parallelism` указано значение 6:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sleep20
spec:
  completions: 3
  parallelism: 6
  template:
    metadata:
      name: sleep20
    spec:
      containers:
```

```

- name: sleep20
  image: python:3.6
  command: ["python",
            "-c",
            "import time; print('started...');
            time.sleep(20); print('done.')"]
  restartPolicy: Never
> Kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
sleep20-1t8sd                       1/1     Running   0           10s
sleep20-sdjb4                       1/1     Running   0           10s
sleep20-wv4jc                       1/1     Running   0           10s

```

Удаление завершенных заданий

По завершении задание остается в памяти, равно как и его поды. Это сделано специально, чтобы вы могли просмотреть журнальные файлы, подключиться к подам и исследовать их. Но обычно после успешного завершения задание становится ненужным. Очистка завершенных заданий и их подов ложится на вас. Проще всего это сделать, удалив объект `job` и все экземпляры подов:

```

> kubectl delete jobs/factroial5
job "factorial5" deleted
> kubectl delete jobs/sleep20
job "sleep20" deleted

```

Планирование регулярных заданий с помощью crontab

Kubernetes поддерживает регулярные задания, которые можно запускать как один раз, так и многократно. Они основаны на стандартной для Unix системе `crontab`, а их описание находится в файле `/etc/crontab`.

В Kubernetes 1.4 они назывались `ScheduledJob`, но в версии 1.5 их переименовали в `CronJob`. Начиная с версии 1.8 ресурс `CronJob` включен в API-сервере по умолчанию, поэтому больше не нужно указывать флаг `-runtime-config`, хотя эта подсистема все еще находится на стадии бета-тестирования. Далее показана конфигурация для ежеминутного запуска заданий, которые будут напоминать вам о необходимости размяться. При планировании `*` можно поменять на `?`:

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: stretch
spec:
  schedule: "*/* * * * *"
  jobTemplate:

```

```

spec:
  template:
    metadata:
      labels:
        name: stretch
spec:
  containers:
  - name: stretch
    image: python
    args:
    - python
    - -c
    - from datetime import datetime; print('{{
Stretch'.format(datetime.now())
    restartPolicy: OnFailure

```

В спецификации пода внизу шаблона задания я добавил метку `name`. Дело в том, что Kubernetes назначает заданиям и их подам имена с произвольными префиксами. Метка помогает легко найти все поды, выполняющие определенное задание, — смотрите следующий вывод командной строки:

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
stretch-1482165720-qm5bj	0/1	ImagePullBackOff	0	1m
stretch-1482165780-bkqjd	0/1	ContainerCreating	0	6s

Как видите, при каждом вызове регулярного задания запускается объект `job` с новым подом:

```
> kubectl get jobs
```

NAME	DESIRED	SUCCESSFUL	AGE
stretch-1482165300	1	1	11m
stretch-1482165360	1	1	10m
stretch-1482165420	1	1	9m
stretch-1482165480	1	1	8m

Когда задание завершается, его поды переходят в состояние `Completed` и их больше нельзя увидеть без флагов `-show-all` или `-a`:

```
> Kubectl get pods -show-all
```

NAME	READY	STATUS	RESTARTS	AGE
stretch-1482165300-g5ps6	0/1	Completed	0	15m
stretch-1482165360-cln08	0/1	Completed	0	14m
stretch-1482165420-n8nzd	0/1	Completed	0	13m
stretch-1482165480-0jq31	0/1	Completed	0	12m

Как обычно, вывод пода заверщенного задания можно просмотреть с помощью команды `logs`:

```
> kubectl logs stretch-1482165300-g5ps6
[2016-12-19 16:35:15.325283] Stretch
```

После того как будет удалено регулярное задание, прекращается планирование новых заданий и удаляются все существующие объекты `job` и созданные ими поды.

Для поиска всех объектов `job`, запущенных конкретным заданием, можно использовать заранее назначенную метку (в данном случае она называется `STRETCH`). Есть возможность приостановить регулярное задание, чтобы оно не создавало новые объекты `job` и поды, пока вы не удалите уже завершенные. Чтобы управлять предыдущими заданиями, в разделе `спес` можно указать поля `successfulJobsHistoryLimit` и `failedJobsHistoryLimit`.

Интеграция с внешними компонентами

Большинство компонентов реального времени в кластере Kubernetes взаимодействуют с внекластерными компонентами. Эти сервисы могут быть как внешними, доступными через некий сторонний API, так и внутренними, размещенными в той же локальной сети, но по различным причинам не входящими в кластер Kubernetes.

Их можно разделить на две категории: внутри сети кластера и вне ее. Почему это различие имеет значение?

Компоненты вне сети кластера

Такие компоненты не имеют прямого доступа к кластеру и обращаются к нему только через API, URL-адреса, видимые извне, и выведенные наружу сервисы. С ними обращаются так же, как с любым внешним пользователем. Часто компоненты кластера сами используют внешние сервисы, что не влечет за собой никаких рисков, связанных с безопасностью. Например, на моей предыдущей работе был кластер Kubernetes, который сообщал об исключениях стороннему сервису (sentry.io/welcome/). Взаимодействие протекало в одном направлении — от кластера Kubernetes к стороннему сервису.

Компоненты внутри сети кластера

Существуют такие компоненты, которые работают в одной сети с кластером, но им не контролируются. У выбора такого решения может быть множество причин. Это и устаревшие приложения, которые еще не успели перенести на Kubernetes, и какое-то распределенное хранилище данных, которое сложно интегрировать с этой платформой. Данные компоненты помещают во внутреннюю сеть для увеличения производительности, а также их изоляции от внешнего мира, чтобы трафик между ними и подами был более безопасным. Размещение в одной и той же сети сокращает время задержек и снижает необходимость в аутентификации, что не только удобно, но и выгодно с точки зрения накладных расходов.

Управление системой Hue с помощью Kubernetes

В этом разделе вы узнаете, каким образом Kubernetes помогает в работе с огромной системой, такой как Hue. Платформа Kubernetes предоставляет множество возможностей для оркестрации подов, управления квотами и лимитами, обнаружения и устранения определенного рода стандартных сбоев, таких как аппаратные неполадки, отказ процесса или недоступный сервис. Однако в сложных системах наподобие Hue состояние вполне рабочих подов и сервисов может оказаться некорректным или они застопорятся в ожидании других зависимостей, необходимых для выполнения их обязанностей. Это нетривиальная ситуация, так как, если еще не готовый под начал принимать запросы, вам нужно как-то на них отреагировать: отклонить, возлагая ответственность на вызывающую сторону, попросить повторить (*сколько раз? как долго? как часто?*) или поместить в очередь ожидания (*кто будет отвечать за эту очередь?*).

Часто имеет смысл позволить системе следить за готовностью различных компонентов или же делать эти компоненты видимыми только тогда, когда они полностью готовы. Платформе Kubernetes ничего не известно о Hue, но она предоставляет несколько механизмов, таких как проверка работоспособности/готовности и контейнеры инициализации, которые позволяют управлять кластером с учетом конкретного приложения.

Проверка работоспособности контейнеров

Kubect1 следит за вашими контейнерами. Если с процессом контейнера произойдет сбой, проблема будет решена с помощью Kubelet и политики перезапуска. Но этого не всегда достаточно. Процесс способен войти в бесконечный цикл или заблокироваться, но при этом оставаться в рабочем состоянии. Политика перезапуска может оказаться недостаточно гибкой. С помощью проверки работоспособности вы сами решаете, «жив» контейнер или нет. Далее представлен шаблон пода для музыкального сервиса Hue. Он содержит раздел `livenessProbe`, в котором значится проверка `HttpGet`. Для HTTP-проверки требуется указать полный адрес: HTTP или HTTPS (по умолчанию используется HTTP), адрес узла (по умолчанию `PodIp`), путь и порт. Проверка считается пройденной, если HTTP-код находится в диапазоне между 200 и 399. Контейнеру может потребоваться некоторое время для инициализации, поэтому укажите значение `initialDelayInSeconds`, Kubelet начнет проверку лишь по истечении этого времени:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: hue-music
    name: hue-music
spec:
  containers:
```

```

image: the_g1g1/hue-music
livenessProbe:
  httpGet:
    path: /pulse
    port: 8888
    httpHeaders:
      - name: X-Custom-Header
        value: ItsAlive
  initialDelaySeconds: 30
  timeoutSeconds: 1
name: hue-music

```

Если хотя бы один контейнер не пройдет проверку работоспособности, применяется политика перезапуска пода. Убедитесь в том, что не используете политику *Never*, иначе проверка окажется бесполезной.

Существует еще два вида проверок:

- ❑ `TcpSocket` — просто проверяет, открыт ли порт;
- ❑ `Exec` — запускает команду, которая в случае успеха возвращает `0`.

Управление зависимостями с помощью проверок готовности

Проверки готовности применяются для разных целей. Контейнер может быть запущен, но при этом зависеть от других сервисов, недоступных в данный момент. Например, компоненту `hue-music` требуется доступ к сервису, который содержит список прослушанных вами композиций. Без этого он не способен выполнять свои обязанности. В данном случае другие сервисы или внешние клиенты должны воздержаться от отправки запросов компоненту `hue-music`, однако необходимости перезапускать его тоже нет. Проверки готовности решают эту проблему. Если контейнер не выдержал такую проверку, его под убирается из всех конечных точек, в которых он зарегистрирован. Благодаря этому запросы не могут перегрузить сервис, который не в состоянии их обработать. С помощью проверки готовности можно также временно отключать поды, для которых в данный момент приготовлено слишком много запросов в какой-нибудь внутренней очереди.

Далее приводится пример проверки готовности. Я использую проверку типа `exec`, чтобы выполнить *собственную* команду. Если команда вернет ненулевой код завершения, контейнер будет удален:

```

readinessProbe:
  exec:
    command:
      - /usr/local/bin/checker
      - --full-check
      - --data-service=hue-multimedia-service
  initialDelaySeconds: 60
  timeoutSeconds: 5

```


Один и тот же контейнер можно проверять как на готовность, так и на работоспособность, поскольку эти проверки имеют разные цели.

Применение контейнеров инициализации для упорядоченного запуска подов

Проверки работоспособности и готовности — отличные инструменты. Они знают, что при запуске контейнер готов не сразу и в это время его не следует считать неисправным. Для определения задержки перед проверкой предусмотрен параметр `initialDelayInSeconds`. Но что, если начальная задержка может оказаться слишком длительной? Для подготовки к обработке запросов большинству контейнеров требуется несколько секунд, но если начальная задержка занимает 5 мин, контейнер будет слишком долго простаивать без работы. И если он является частью сильно нагруженного сервиса, это может вызвать простаивание множества узлов после каждого обновления, что сделает сервис практически недоступным.

Эту проблему решают контейнеры инициализации. Они запускаются внутри одного пода и завершаются до того, как будут запущены остальные контейнеры. Они берут на себя процесс инициализации, который занимает неопределенное время, и позволяют обычным контейнерам запускаться с минимальной задержкой, чтобы они могли пройти проверку готовности.

В Kubernetes 1.6 контейнеры инициализации вышли из состояния бета. Их можно указать в спецификации пода в поле `initContainers` по аналогии с полем `containers`. Далее приведен пример:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-fitness
spec:
  containers:
    name: hue-fitness
    Image: hue-fitness:v4.4
  InitContainers:
    name: install
    Image: busybox
    command: /support/safe_init
    volumeMounts:
      - name: workdir
        mountPath: /workdir
```

Предоставление данных для подов DaemonSet. Поды DaemonSet развертываются автоматически по одному на каждом узле или определенном подмножестве узлов. Обычно они следят за узлами и их работоспособностью. Мы уже затрагивали эту крайне важную функцию при обсуждении сервиса Node Problem Detector в главе 3. Но этим их возможности не ограничиваются. Стандартный планировщик Kubernetes развертывает поды в зависимости от нагрузки и до-

ступности ресурсов. Если у вас есть множество не слишком ресурсоемких подов, значительная их часть окажется на одном узле. Рассмотрим под, который выполняет небольшую задачу и затем каждую секунду передает удаленному сервису отчет обо всех своих действиях. Представьте, что в среднем 50 таких подов развертываются на одном и том же узле. Это означает, что каждую секунду будут выполняться 50 сетевых запросов с незначительным количеством данных. Но что, если мы уменьшим их количество в 50 раз, ограничившись лишь одним сетевым запросом? Вместо того чтобы взаимодействовать непосредственно с внешним сервисом, поды могут обращаться к **DaemonSet**. **DaemonSet** будет собирать все данные из всех 50 подов и, объединив, каждую секунду отправлять их наружу. Конечно, API удаленного сервиса должен поддерживать ответы в таком формате. Хорошая новость состоит в том, что вам не нужно модифицировать сами поды — они просто будут сконфигурированы для обращения к локальному экземпляру **DaemonSet** вместо удаленного сервиса. В данном случае под с названием **DaemonSet** играет роль агрегирующего прокси-сервера.

Интересная особенность этого конфигурационного файла — то, что поля **hostNetwork**, **hostPID** и **hostIPC** равны **true**. Это позволяет подам эффективно взаимодействовать с прокси, пользуясь тем фактом, что все они находятся на одном физическом узле:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hue-collect-proxy
  labels:
    tier: stats
    app: hue-collect-proxy
spec:
  template:
    metadata:
      labels:
        hue-collect-proxy
    spec:
      hostPID: true
      hostIPC: true
      hostNetwork: true
      containers:
        image: the_g1g1/hue-collect-proxy
        name: hue-collect-proxy
```

Развитие системы Hue с помощью Kubernetes

В этом разделе мы обсудим другие способы расширения Hue и обслуживания дополнительных рынков и сообществ. Вопрос остается прежним: *какие особенности и возможности Kubernetes помогут нам справиться с новыми вызовами и требованиями?*

Применение Hue на предприятиях

Многие предприятия не способны использовать облака. Это может быть продиктовано как требованиями к безопасности и соблюдению норм законов, так и соображениями производительности, если системе необходимо работать с данными и уже существующими приложениями, которые нецелесообразно переносить в облако. Как бы то ни было, система Hue должна поддерживать кластеры, размещенные локально и/или работающие на «голом железе».

Платформа Kubernetes чаще всего развертывается в облаке, у нее даже есть для этого специальный интерфейс `cloud-provider`, но на самом деле ее можно развернуть где угодно. Потребуется некоторый опыт, но у промышленных организаций, которые уже обслуживают системы в собственных вычислительных центрах, с этим не должно быть никаких проблем.

CoreOS предоставляет много материала о развертывании кластеров Kubernetes на «голом железе».

Двигаем научный прогресс с помощью Hue

Система Hue так хорошо интегрирует информацию из разных источников, что ее применение в сфере науки стало бы настоящей находкой. Только подумайте, насколько она могла бы упростить сотрудничество между учеными, работающими в разных областях.

Для поддержки множества научных учреждений может потребоваться развертывание географически распределенных кластеров или даже многокластерный режим. Такой сценарий использования не чужд для платформы Kubernetes, и его поддержка постоянно эволюционирует. Мы подробно обсудим эту тему в одной из последующих глав.

Hue — будущее системы образования

Hue можно применять в учебных заведениях и для предоставления услуг в сфере интернет-образования. Но опасения относительно сохранности личных данных могут стать преградой на пути развертывания Hue в виде единой централизованной системы. Как вариант, единый кластер можно разделить на пространства имен для разных школ. Или развернуть в каждом учебном заведении или округе отдельный кластер Hue. В таком случае система должна быть очень простой в управлении и обслуживании, чтобы ею могли пользоваться даже в школах, где нет опытных специалистов. Существенную помощь в этом способна оказать платформа Kubernetes, благодаря которой у системы Hue есть механизмы самовосстановления и автоматического масштабирования, что максимально упрощает ее администрирование.

Резюме

В этой главе мы спроектировали и запланировали разработку, развертывание и администрирование всеведущей и всеильной (пусть только в воображении) системы Hue, построенной на основе микросервисов. Естественно, в качестве базовой платформы для оркестрации мы задействовали Kubernetes, что позволило углубиться во множество ее концепций и ресурсов. В частности, был сделан акцент на развертывании подов для «долгоиграющих» сервисов (вместо заданий для запуска краткосрочных или регулярных процессов). Мы также показали, чем внутренние сервисы отличаются от внешних, и воспользовались пространствами имен для сегментирования кластера Kubernetes. Затем перешли к теме управления крупными системами наподобие Hue с использованием проверок работоспособности и готовности, контейнеров инициализации и `DaemonSet`.

Теперь у вас не должно возникнуть проблем с масштабируемыми веб-системами, состоящими из микросервисов, равно как и с их развертыванием и администрированием в кластере Kubernetes.

Следующая глава посвящена крайне важной области — хранению данных. Данные являются ключевым, но, как часто бывает, наименее гибким элементом системы. Kubernetes предоставляет модель хранения и множество механизмов для интеграции с различными базами данных.

7

Работа с хранилищем данных в Kubernetes

В этой главе вы разберетесь, как Kubernetes работает с хранилищами данных. Хранение и вычисление — две совершенно разные области, но в целом они касаются ресурсов. Kubernetes как универсальная платформа старается абстрагировать хранилище с помощью программных моделей и набора дополнений для различных баз данных. Сначала мы углубимся в подробности концептуальной модели хранения и предоставления доступа к хранилищу для контейнеров в кластере. Затем рассмотрим хранилища данных в популярных облачных провайдерах, таких как AWS, GCE и Azure. Мы также уделим внимание известному провайдеру с открытым исходным кодом (GlusterFS от Red Hat), который предоставляет распределенную файловую систему. Вашему вниманию будет представлено альтернативное решение — Flocker, которое управляет данными в контейнерах и является частью кластера Kubernetes. В конце вы узнаете, каким образом Kubernetes обеспечивает интеграцию существующих хранилищ.

По прочтении этой главы вы получите четкое понимание о представлении хранилищ в Kubernetes. Будете знать, какие методы хранения доступны в той или иной среде развертывания — тестовой, облачной и промышленной, и сможете выбрать тот, который лучше всего подходит для вашего случая.

Подробное знакомство с постоянными томами

В этом разделе мы рассмотрим концептуальную модель хранения данных в Kubernetes и попробуем подключить постоянное хранилище к контейнерам, чтобы они могли читать и изменять его содержимое. Для начала поговорим о проблемах, связанных с хранением. Время жизни контейнеров и подов ограничено. Все, что контейнер записывает в своей файловой системе, теряется при его удалении. Контейнеры также способны подключать для чтения/записи каталоги своего узла. Это позволит сохранить данные в случае их перезапуска, однако сами узлы не вечны.

Существуют и другие проблемы, например: что делать с правами на владение подключенными каталогами узла после удаления контейнера? Представьте, что множество контейнеров записывают важные данные в разные участки файловой системы своих узлов. Но после их удаления невозможно будет сказать, кому принадлежат те или иные данные. Вы можете попытаться сохранять эту информацию, но куда? Очевидно, что для надежного управления данными в крупномасштабных системах необходимо постоянное хранилище, доступное из любого узла.

Томы

Базовая абстракция хранения данных в Kubernetes — это том. Контейнеры подключают тома, привязанные к их подам, и обращаются к хранилищу (чем бы оно ни было) как к своей локальной файловой системе. Это отличный, проверенный временем метод, благодаря которому разработчику приложения не нужно беспокоиться о том, где и как хранятся его данные.

Использование `emptyDir` для взаимодействия внутри пода

Контейнеры в одном поде могут легко обмениваться данными с помощью общего (разделяемого) тома. Они просто подключают один и тот же том и общаются друг с другом путем записи и чтения в общем пространстве. Самый простой том — `emptyDir`. Это *пустой* каталог, принадлежащий узлу, он не постоянный, поскольку его содержимое очищается при удалении пода. Если с контейнером произойдет сбой, под останется на месте, и позже вы сможете обратиться к тому же тому. Еще один интересный вариант — применение RAM-диска. Для этого в качестве носителя нужно указать `Memory`. Таким образом, ваши контейнеры взаимодействуют через оперативную память. Данный способ более быстрый, но, конечно же, не очень надежный. Если узел перезагрузится, содержимое тома `emptyDir` будет утеряно.

Далее представлен конфигурационный файл пода с двумя контейнерами, подключающими один и тот же том `shared-volume`. Они используют для этого разные пути, но файл, записанный контейнером `hue-global-listener` в каталог `/notifications`, становится доступным в каталоге `/incoming` контейнера `hue-job-scheduler`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-scheduler
spec:
  containers:
    - image: the_g1g1/hue-global-listener
      name: hue-global-listener
      volumeMounts:
        - mountPath: /notifications
          name: shared-volume
    - image: the_g1g1/hue-job-scheduler
      name: hue-job-scheduler
      volumeMounts:
        - mountPath: /incoming
          name: shared-volume
  volumes:
    - name: shared-volume
      emptyDir: {}
```

Чтобы использовать разделяемую память, следует в раздел `emptyDir` добавить параметр `medium: Memory`:

```
volumes:
- name: shared-volume
  emptyDir:
    medium: Memory
```

Использование HostPath для взаимодействия внутри пода

Иногда имеет смысл открыть подам доступ к системной информации (например, о Docker Daemon) или позволить им взаимодействовать между собой. Это может быть полезно в том случае, если подам известно, что они находятся на одном узле. Но обычно этих сведений у них нет, так как Kubernetes разворачивает их в зависимости от доступных ресурсов.

Есть две ситуации, когда поды могут рассчитывать на то, что все они делят один и тот же узел.

- ❑ В одноузловом кластере все поды, естественно, размещаются на одном узле.
- ❑ Поды DaemonSet, у которых совпадает селектор, всегда принадлежат одному и тому же узлу.

В главе 6 мы рассматривали под DaemonSet, который играет роль агрегирующего прокси-сервера для других подов. Для альтернативной реализации этой идеи можно было бы заставить поды записывать свои данные в подключенный том, привязанный к каталогу `host`. В таком случае DaemonSet будет реагировать на эти данные, считывая их напрямую.

Прежде чем использовать том HostPath, ознакомьтесь с его ограничениями.

- ❑ Поведение подов с одинаковой конфигурацией может различаться, если их работа зависит от данных и при этом файлы на их узле различны.
- ❑ Том способен нарушить планирование, основанное на ресурсах (скоро появится в Kubernetes), поскольку у Kubernetes нет возможности следить за ресурсами HostPath.
- ❑ У контейнеров, которые обращаются к каталогам узла, должен быть контекст безопасности с параметром `privileged`, равным `true`. Того же эффекта можно добиться, открыв доступ на запись на стороне узла.

Далее показан конфигурационный файл, который подключает к контейнеру `hue-coupon-hunter` каталог `/coupons`, привязанный к каталогу узла `/etc/hue/data/coupons`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-coupon-hunter
```

```

spec:
  containers:
  - image: the_g1g1/hue-coupon-hunter
    name: hue-coupon-hunter
    volumeMounts:
    - mountPath: /coupons
      name: coupons-volume
  volumes:
  - name: coupons-volume
    host-path:
      path: /etc/hue/data/coupons

```

Так как у пода нет контекста безопасности `privileged`, он не сможет записывать в каталог `host`. Чтобы это исправить, добавим контекст безопасности в спецификацию контейнера:

```

- image: the_g1g1/hue-coupon-hunter
  name: hue-coupon-hunter
  volumeMounts:
  - mountPath: /coupons
    name: coupons-volume
  securityContext:
    privileged: true

```

Как показано на рис. 7.1, у каждого контейнера есть собственное локальное хранилище, недоступное другим контейнерам и подам. При этом к обоим контейнерам подключается каталог `/data`, принадлежащий узлу.

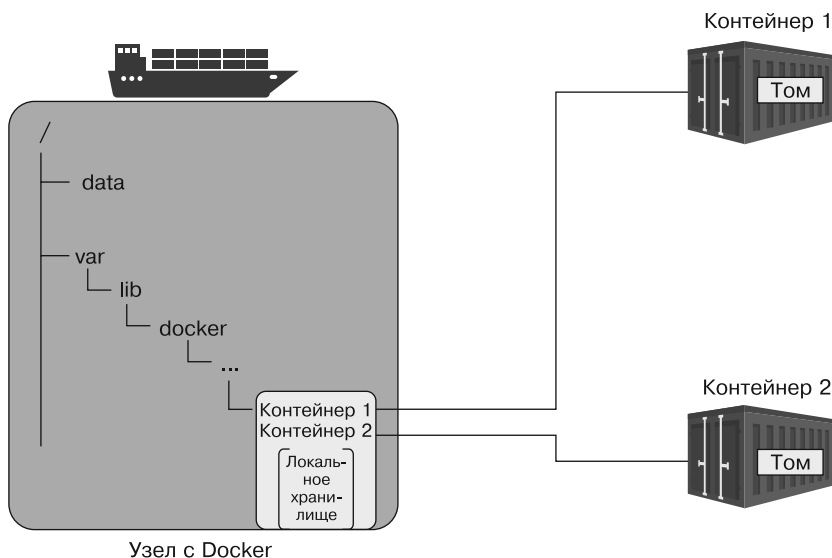


Рис. 7.1

Постоянное хранилище для узла на основе локальных томов

Локальные тома похожи на `HostPath`, но они не удаляются во время перезапуска пода или узла. В этом смысле их можно считать постоянными. Они появились в Kubernetes 1.7, но, начиная с версии 1.10, для их включения нужно устанавливать флаг `--feature-gates`. Локальные тома предназначены для поддержки объектов `StatefulSet`, которые позволяют развертывать определенные поды на узлах с определенными томами. Локальные тома предоставляют аннотации принадлежности к узлу, что упрощает привязку подов к нужным им хранилищам:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
  annotations:
    "volume.alpha.kubernetes.io/node-affinity": '{
      "requiredDuringSchedulingIgnoredDuringExecution": {
        "nodeSelectorTerms": [
          { "matchExpressions": [
            { "key": "kubernetes.io/hostname",
              "operator": "In",
              "values": ["example-node"]
            }
          ]
        }
      }
    }'
spec:
  capacity:
    storage: 100Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
```

Выделение постоянных томов

Тома типа `emptyDir` можно подключать и использовать внутри контейнеров, но они не постоянные и не требуют специального выделения, так как в их основе лежит уже существующее хранилище узла. Тома `HostPath` постоянны в рамках одного узла, но если под будет перезапущен на другом узле, он не получит доступ к тому, с которым работал ранее. Тома `Local` тоже постоянны на уровне узла и могут пережить повторное развертывание и перезапуск подов и даже самих узлов. По-настоящему постоянный том вместо диска, физически подключенного к узлу, задействует внешнее хранилище, которое заблаговременно выделяют администраторы. В облачных средах выделение может быть автоматизировано, но без него все

равно не обойтись. Как администратор кластера Kubernetes, вы должны убедиться в том, что хранилище имеет подходящие квоты, и тщательно отслеживать использование доступного пространства.

Помните, что постоянные тома — это ресурсы, которые в Kubernetes применяются по аналогии с узлами. В связи с этим они не управляются API-сервером. Ресурсы можно выделять статически и динамически.

- ❑ **Статическое выделение постоянных томов.** Это простая операция. Администратор кластера заблаговременно создает постоянные тома на основе какого-то носителя данных, после чего их могут подключать разные контейнеры.
- ❑ **Динамическое выделение постоянных томов.** Может происходить в момент, когда ни один из статических томов не соответствует запросу контейнера. Если в запросе указан класс хранилища, сконфигурированный администратором для динамического выделения, постоянный том может быть выделен на лету. Мы обсудим классы хранилищ и запрашивание постоянных томов в приведенных далее примерах.
- ❑ **Внешнее выделение постоянных томов.** В последнее время команда Kubernetes старается вынести механизм выделения хранилищ из ядра платформы и оформить его в виде подключаемых (сторонних) томов. Внешний механизм работает по аналогии с внутренним динамическим выделением, но его можно разворачивать и обновлять отдельно. Все больше подсистем выделения выносятся из основной ветки. Взгляните на проект Kubernetes incubator: github.com/kubernetes-incubator/external-storage.

Создание постоянных томов

Конфигурационный файл для постоянного тома на основе NFS выглядит следующим образом:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-1
  labels:
    release: stable
    capacity: 100Gi
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: normal
  nfs:
    path: /tmp
    server: 172.17.0.8
```

У постоянного тома есть спецификация и метаданные, где указывается его имя. Сосредоточимся на разделе `spec`. Он состоит из следующих подразделов: емкость (`capacity`), режим тома (`volumeMode`), режимы доступа (`accessModes`), политика удаления (`persistentVolumeReclaimPolicy`), класс хранилища (`storageClassName`) и тип тома (в данном случае `nfs`).

Емкость

У каждого тома есть определенное доступное пространство. Запрос на выделение хранилища (`claim`) может быть удовлетворен томами, у которых есть необходимое место. В приведенном примере объем тома — 100 Гбайт (230 байт). При статическом выделении постоянных томов необходимо понимать принцип, по которому они запрашиваются. Представьте: вы выделили 20 постоянных томов по 100 Гбайт каждый, а контейнер запрашивает постоянный том размером 150 Гбайт. Этот запрос не будет удовлетворен, хотя общей емкости вполне достаточно:

```
capacity:
  storage: 100Gi
```

Режим тома

В Kubernetes 1.9 в состоянии альфа была добавлена опциональная возможность указывать режим тома для статического выделения (хотя это делается в разделе `spec`, а не в виде аннотации). Это позволяет выбирать между файловой системой ("`Filesystem`") и блочным хранилищем ("`Block`"). Как и в версиях, предшествовавших 1.9, по умолчанию используется режим "`Filesystem`".

Режимы доступа

Существует три режима доступа:

- ☐ `ReadOnlyMany` — можно подключать к нескольким узлам в режиме чтения;
- ☐ `ReadWriteOnce` — можно подключать к одному узлу в режиме чтения и записи;
- ☐ `ReadWriteOnce` — можно подключать к нескольким узлам в режиме чтения и записи.

Хранилище подключается к узлу, поэтому даже в режиме `ReadWriteOnce` оно может быть подключено на запись несколькими контейнерами на одном и том же узле. Если с этим возникнут проблемы, нужно будет решить их через какой-то другой механизм (например, том запрашивается только из подов `DaemonSet`, которые существуют на каждом узле в единственном экземпляре).

Некоторые провайдеры хранилищ поддерживают эти режимы не полностью. При выделении постоянного тома можно указать, в каких режимах он будет работать. Например, NFS поддерживает любой режим, но в приведенной далее спецификации доступны только два:

```
accessModes:
  - ReadWriteMany
  - ReadOnlyMany
```

Политика удаления

Политика удаления определяет, что происходит при исчезновении запроса на выделение постоянного тома. Существует три варианта:

- ❑ **Retain** — том нужно удалить вручную;
- ❑ **Delete** — соответствующее хранилище, такое как AWS EBS, GCE PD, диск Azure или том OpenStack Cinder, удаляется;
- ❑ **Recycle** — удаляется только содержимое (`rm -rf /volume/*`).

Выбор политик **Retain** или **Delete** означает, что постоянный том окажется недоступен для последующих запросов. Политика **Recycle** позволяет запросить том повторно.

В настоящий момент **Recycle** поддерживается только в NFS и HostPath. AWS EBS, GCE PD, диск Azure и тома Cinder поддерживают удаление. Динамически выделяемые тома удаляются всегда.

Класс хранилища

Вы можете указать класс хранилища с помощью поля `storageClassName` в разделе `spec`. В этом случае к постоянному тому будут поступать только запросы с подходящим классом хранилища. Если опустить это поле, для запросов тома не обязательно указывать класс.

Тип тома

Тип тома указывается по названию в разделе `spec`. Подраздела `volumeType` не существует. В предыдущем примере в качестве типа тома значилось `nfs`:

```
nfs:
  path: /tmp
  server: 172.17.0.8
```

У каждого типа тома может быть свой набор параметров. В данном случае это `path` и `server`.

Позже в этой главе мы пройдемся по разным типам.

Запрос постоянного тома

Когда контейнер хочет получить доступ к какому-нибудь постоянному хранилищу, он выполняет запрос (или, скорее, разработчик и администратор договариваются о необходимом объеме ресурсов). Далее приводится пример запроса, который соответствует постоянному тому из предыдущего раздела:

```
kind: PersistentVolumeClaim
apiVersion: v1
```

```

metadata:
  name: storage-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 80Gi
  storageClassName: "normal"
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: capacity, operator: In, values: [80Gi, 100Gi]}

```

Имя `storage-claim` пригодится позже, при подключении запроса к контейнеру.

В спецификации указан режим доступа `ReadWriteOnce`. Это означает, что после удовлетворения данного запроса любой другой запрос в этом режиме будет отклонен, хотя режим `ReadOnlyMany` по-прежнему доступен.

В разделе ресурсов запрашивается 80 Гбайт. Этот запрос может быть удовлетворен постоянным томом емкостью 100 Гбайт, но 20 Гбайт останутся незадействованными, что немного расточительно.

Класс хранилища определен как `"normal"`. Мы уже упоминали, что он должен совпадать с классом постоянного тома. Однако с точки зрения формата PVC (`Persistent Volume Claim` — запрос постоянного тома) пустое имя класса (`""`) — это не то же самое, что отсутствие класса как такового. Первое соответствует постоянным томам без класса хранилища, второе позволяет подключать постоянные тома только при выключенном дополнении `DefaultStorageClass` (регулирует доступ) или если оно включено и при этом используется класс хранилища по умолчанию.

Раздел `Selector` позволяет дополнительно отфильтровать доступные тома. Например, в нашем случае у тома должно быть две метки: одна — `release: "stable"`, а другая — либо `capacity: 80 Gi`, либо `capacity: 100 Gi`. Представьте, что выделено еще несколько томов по 200 и 500 Гбайт. Мы не хотим запрашивать том емкостью 500 Гбайт, если нужно всего 80 Гбайт.

Kubernetes всегда пытается выбрать наименьший том, способный удовлетворить запрос, но если томов емкостью 80 или 100 Гбайт просто нет, а тома по 200 и 500 Гбайт нельзя назначить из-за меток, место будет выделено динамически.

Важно понимать, что запросы не содержат имена томов. Kubernetes делает выбор на основе класса хранилища, емкости и меток.

Наконец, запрос постоянного тома принадлежит определенному пространству имен. Сопоставление постоянного тома и запроса эксклюзивно. Это означает, что постоянный том будет привязан к пространству имен, в него должны входить любые поды, подключающие данный том, даже если используются режимы доступа `ReadOnlyMany` или `ReadWriteMany`.

Подключение запросов в качестве томов

Итак, мы выделили и запросили том. Пришло время воспользоваться запрошенным пространством в контейнере. Это довольно просто. Запрос должен быть представлен в виде тома в поде, после чего его контейнеры смогут его подключать как любой другой том. Далее представлен конфигурационный под-файл, в котором указан запрос постоянного тома, созданный ранее (и привязанный к постоянному тому на основе NFS, который мы выделили):

```
kind: Pod
apiVersion: v1
metadata:
  name: the-pod
spec:
  containers:
  - name: the-container
    image: some-image
    volumeMounts:
    - mountPath: "/mnt/data"
      name: persistent-volume
  volumes:
  - name: persistent-volume
    persistentVolumeClaim:
      claimName: storage-claim
```

Ключевой момент здесь — подраздел `persistentVolumeClaim` в разделе `volumes`. Имя запроса (в данном случае `storage-claim`) уникально в рамках текущего пространства имен, оно делает его доступным в виде тома `persistent-volume`. Теперь контейнер может обратиться к нему по имени и подключить его к `/mnt/data`.

Томы на основе блочных устройств

Возможность использовать томы на основе блочных устройств появилась в Kubernetes 1.9 в виде альфа. Включается она с помощью параметра `--feature-gates=BlockVolume=true`.

Блочные томы предоставляют прямой доступ к исходному хранилищу, без прослойки в виде файловой системы. Это чрезвычайно полезно для приложений, требующих от хранилища высокой производительности (например, базы данных) или стабильной скорости ввода/вывода и низкого уровня задержек. В качестве блочного хранилища используются Fiber Channel, iSCSI и локальные SSD. В настоящий момент (Kubernetes 1.10) блочные томы поддерживаются провайдерами хранилищ `Local Volume` и `FiberChannel`. Блочный том определяется следующим образом:

```
apiVersion: v1
kind: PersistentVolume
```

```

metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
volumeMode: Block
persistentVolumeReclaimPolicy: Retain
fc:
  targetWWNs: ["50060e801049cfd1"]
  lun: 0
  readOnly: false

```

В соответствующем запросе PVC тоже должен быть параметр `volumeMode: Block`. Вот как это может выглядеть:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
  requests:
    storage: 10Gi

```

Под подключает блочный том в виде устройства в каталоге `/dev`, а не как файловую систему. Позже контейнеры смогут обращаться к данному устройству для чтения/записи. На практике это означает, что запросы ввода/вывода передаются непосредственно блочному хранилищу, минуя драйвер файловой системы. Теоретически это должно повысить производительность, но в реальности результат может оказаться противоположным, если приложение полагается на буферизацию файловой системы.

Далее представлен под с контейнером, который привязывает `block-pvc` к блочному хранилищу в виде устройства с именем `/dev/xdva`:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]

```

```

    volumeDevices:
      - name: data
        devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc

```

Классы хранилищ

Классы хранилищ позволяют администратору подключать к кластеру нестандартные постоянные носители (если только для их поддержки есть подходящие дополнения). Класс хранилища имеет поля `provisioner` и `parameters`, а также `name` в разделе `metadata`:

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2

```

Для одного и того же компонента, выделяющего дисковое пространство, можно создать несколько классов хранилищ. Каждый компонент имеет отдельный набор параметров.

Сейчас поддерживаются следующие типы томов:

- ☐ `AwsElasticBlockStore;`
- ☐ `AzureFile;`
- ☐ `AzureDisk;`
- ☐ `CephFS;`
- ☐ `Cinder;`
- ☐ `FC;`
- ☐ `FlexVolume;`
- ☐ `Flocker;`
- ☐ `GcePersistentDisk;`
- ☐ `GlusterFS;`
- ☐ `ISCSI;`
- ☐ `PhotonPersistentDisk;`
- ☐ `Quobyte;`
- ☐ `NFS;`
- ☐ `RBD;`

- ❑ VsphereVolume;
- ❑ PortworxVolume;
- ❑ ScaleIO;
- ❑ StorageOS;
- ❑ Local.

В список не вошли типы томов `gitRepo` и `secret`, не основанные на традиционных сетевых хранилищах. Этот аспект Kubernetes все еще активно развивается, и позже мы увидим дальнейшее разделение компонентов и более аккуратную архитектуру, в которой дополнения перестанут быть частью проекта Kubernetes. Разумный выбор томов подходящего типа — важная часть проектирования и администрирования кластера.

Класс хранилища по умолчанию. Администратор кластера может назначить класс хранилища, который будет использоваться по умолчанию при динамическом выделении места по запросам, в которых не указано никакого класса. В этом случае необходимо включить дополнение `DefaultStorageClass`. Если такой класс не определен или не включено дополнение `DefaultStorageClass`, запросы без класса хранилища будут соответствовать лишь томам, у которых тоже нет поля `storage`.

Пример работы с постоянным томом от начала до конца

Чтобы проиллюстрировать все рассмотренные ранее концепции, приведу в качестве небольшого примера процесс создания, запрашивания и подключения тома `HostPath`, и затем выполним запись в него из контейнеров.

Для начала создадим том `hostPath`. Сохраните следующий файл с именем `persistent-volume.yaml`:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: persistent-volume-1
spec:
  StorageClassName: dir
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

```
> kubectl create -f persistent-volume.yaml
persistentvolume "persistent-volume-1" created
```

Для просмотра списка доступных томов можно воспользоваться типом ресурсов `persistentvolumes` (сокращенно `pv`):

```
> kubectl get pv
NAME:          persistent-volume-1
CAPACITY:      1Gi
ACCESS MODES:  RWO
RECLAIM POLICY: Retain
STATUS:        Available
CLAIM:
STORAGECLASS:  dir
REASON:
AGE:           17s
```

Я немного отредактировал вывод, чтобы его было легче прочитать. Емкость равна 1 Гбайт, как и требовалось. Здесь указана политика удаления `Retain`, поскольку тома `HostPath` сохраняются. Состояние равно `Available`, так как том еще не был запрошен. В качестве режима доступа выбран `RWX` (сокращение от `ReadWriteMany`). Каждый режим доступа имеет сокращенное название:

- ☐ `ReadWriteOnce` — `RWO`;
- ☐ `ReadOnlyMany` — `ROX`;
- ☐ `ReadWriteMany` — `RWX`.

Постоянный том готов. Теперь создадим запрос. Сохраните следующий файл с названием `persistent-volume-claim.yaml`:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: persistent-volume-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Затем запустите такую команду:

```
> kubectl create -f persistent-volume-claim.yaml
persistentvolumeclaim "persistent-volume-claim" created
```

Проверим запрос и том:

```
> kubectl get pvc
NAME                                STATUS VOLUME
CAPACITY  ACCESSMODES  AGE      persistent-volume-1  1Gi      RWO
persistent-volume-claim  Bound
dir                                1m
```

```
> kubectl get pv
NAME:                persistent-volume-1
CAPACITY:            1Gi
ACCESS MODES:        RWO
RECLAIM POLICY:      Retain
STATUS:              Bound
CLAIM:               default/persistent-volume-claim
STORAGECLASS:        dir
REASON:
AGE:                 3m
```

Как видите, запрос и том привязаны друг к другу. Заключительными шагами будут создание пода и назначение ему запроса в виде тома. Сохраните следующий код в файл `shell-pod.yaml`:

```
kind: Pod
apiVersion: v1
metadata:
  name: just-a-shell
  labels:
    name: just-a-shell
spec:
  containers:
    - name: a-shell
      image: ubuntu
      command: ["/bin/bash", "-c", "while true ; do sleep 10 ; done"]
      volumeMounts:
        - mountPath: "/data"
          name: pv
    - name: another-shell
      image: ubuntu
      command: ["/bin/bash", "-c", "while true ; do sleep 10 ; done"]
      volumeMounts:
        - mountPath: "/data"
          name: pv
  volumes:
    - name: pv
      persistentVolumeClaim:
        claimName: persistent-volume-claim
```

Данный под содержит два контейнера, которые используют образ Ubuntu и запускают консольную команду, просто бездействующую в бесконечном цикле. Это сделано для того, чтобы контейнеры продолжали работать, пока мы к ним подключаемся и проверяем их файловые системы. Под подключает наш запрос постоянного тома под именем `pv`. Оба контейнера подключают итоговый том к каталогу `/data`.

Создадим под и убедимся в том, что оба контейнера запущены:

```
> kubectl create -f shell-pod.yaml
pod "just-a-shell" created
```

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
just-a-shell	2/2	Running	0	1m

Теперь подсоединимся к узлу по ssh. На этом узле каталог `/tmp/data` соответствует тому пода, который подключается к каждому из контейнеров в виде `/data`:

```
> minikube ssh
$
```

Для взаимодействия с контейнерами внутри узла можно использовать команды Docker. Выведем два последних запущенных контейнера:

```
$ docker ps -n 2 --format '{{.ID}}\t{{.Image}}\t{{.Command}}'
820fc954fb96    ubuntu    "/bin/bash -c 'whi..."
cf4502f14be5    ubuntu    "/bin/bash -c 'whi..."
```

Теперь создадим файл в каталоге узла `/tmp/data`. Он должен быть доступен в контейнерах через подключенный том:

```
$ sudo touch /tmp/data/1.txt
```

Выполним консольную команду в одном из контейнеров, чтобы проверить наличие файла, и создадим еще один файл, `2.txt`:

```
$ docker exec -it 820fc954fb96 /bin/bash
root@just-a-shell:/# ls /data
1.txt
root@just-a-shell:/# touch /data/2.txt
root@just-a-shell:/# exit
Finally, we can run a shell on the other container and verify that both
1.txt and 2.txt are visible:
docker@minikube:~$ docker exec -it cf4502f14be5 /bin/bash
root@just-a-shell:/# ls /data
1.txt 2.txt
```

Типы томов в облачных хранилищах GCE, AWS и Azure

В этом разделе мы рассмотрим наиболее распространенные типы томов, доступные на крупнейших публичных облачных платформах. Управление масштабными хранилищами данных — сложная задача, которая в итоге сводится к работе с физическими ресурсами наподобие узлов. Если вы решили разместить свой кластер Kubernetes в публичном облаке, то его провайдер может взять на себя все эти хлопоты, позволяя вам сосредоточиться на своей системе. Но при этом важно разбираться в параметрах и ограничениях, присущих томам того или иного типа.

AWS Elastic Block Store (EBS)

AWS предоставляет для своих серверов EC2 постоянное хранилище с названием EBS. Его можно использовать в кластере Kubernetes, но со следующими ограничениями.

- ❑ Поды должны работать на узлах AWS EC2.
- ❑ Поды могут обращаться только к томам EBS, находящимся в их зоне доступности.
- ❑ Том EBS можно подключить лишь к одному экземпляру EC2.

Это серьезные ограничения. Возможность работать лишь в одной зоне доступности обеспечивает отличную производительность, но не позволяет разделять хранилище между большим количеством участников или распределять его географически — для этого потребуются самостоятельно реализовать репликацию и синхронизацию. Жесткая привязка одного тома EBS к одному экземпляру EC2 означает, что поды не способны разделять хранилище даже внутри одной зоны доступности (и даже для чтения), если их не разместить на одном узле.

Учитывая все эти оговорки, посмотрим, как подключается том EBS:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - image: some-container
    name: some-container
    volumeMounts:
    - mountPath: /ebs
      name: some-volume
  volumes:
  - name: some-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Том EBS создается в AWS, после чего можно подключить его к поду. При этом не нужно создавать запрос и указывать класс хранилища, поскольку подключение происходит напрямую по идентификатору. Тип тома `awsElasticBlockStore` поддерживается в Kubernetes.

AWS Elastic File System

Компания Amazon недавно выпустила сервис под названием *EFS* (Elastic File System — эластичная файловая система). Это управляемое решение на основе NFS.

Оно работает с протоколом NFS 4.1 и имеет множество преимуществ по сравнению с EBS.

- ❑ Разные экземпляры EC2 могут обращаться к одним и тем же файлам, находясь в разных зонах доступности в рамках одного региона.
- ❑ Емкость автоматически масштабируется в зависимости от того, что реально используется.
- ❑ Вы платите только за то, что используете.
- ❑ Вы можете подключать к EFS локальные серверы по VPN.
- ❑ EFS работает поверх SSD-накопителей, которые автоматически реплицируются между зонами доступности.

Однако стоит отметить, что сервис EFS обойдется дороже, чем EBS, даже с учетом автоматической репликации в разных зонах доступности в случае, если вы полностью задействуете свои тома EBS. Он применяет внешний механизм выделения, который непросто развернуть. Следуйте инструкциям, приведенным на github.com/kubernetes-incubator/external-storage/tree/master/aws/efs.

Определив класс хранилища и подготовив постоянный том, вы можете создать запрос и подключить его в режиме `ReadWriteMany` к произвольному количеству подов. Вот как выглядит запрос постоянного тома:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: efs
  annotations:
    volume.beta.kubernetes.io/storage-class: "aws-efs"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

А это под, который его использует:

```
kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
    - name: test-pod
      image: gcr.io/google_containers/busybox:1.24
      command:
        - "/bin/sh"
      args:
        - "-c"
```

```
- "touch /mnt/SUCCESS exit 0 || exit 1"
volumeMounts:
  - name: efs-pvc
    mountPath: "/mnt"
restartPolicy: "Never"
volumes:
  - name: efs-pvc
    persistentVolumeClaim:
      claimName: efs
```

Постоянный диск в GCE

Тип тома `gcePersistentDisk` очень похож на `awsElasticBlockStore`. Диск должен быть выделен заблаговременно, и экземпляры GCE, которые его используют, должны находиться в одном проекте и одной зоне. Однако в режиме чтения он доступен для разных узлов. То есть постоянный диск GCE поддерживает режимы `ReadWriteOnce` и `ReadOnlyMany`. Он применяется для чтения данных из разных подов в рамках одной зоны.

Под, использующий постоянный диск в режиме `ReadWriteOnce`, должен управляться контроллером репликации, набором реплик или быть развернутым с количеством реплик 0 или 1 — дальнейшее масштабирование по понятным причинам невозможно:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
    - image: some-container
      name: some-container
      volumeMounts:
        - mountPath: /pd
          name: some-volume
  volumes:
    - name: some-volume
      gcePersistentDisk:
        pdName: <persistent disk name>
        fsType: ext4
```

Диски в Azure

Azure data disk — это виртуальный жесткий диск, размещенный в хранилище Azure. По своим возможностям он похож на AWS EBS. Далее приведен пример конфигурации пода:

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: some-pod
spec:
  containers:
  - image: some-container
    name: some-container
    volumeMounts:
    - name: some-volume
      mountPath: /azure
  volumes:
  - name: some-volume
    azureDisk:
      diskName: test.vhd
      diskURI: https://someaccount.blob.microsoft.net/vhds/test.vhd

```

Помимо обязательных параметров `diskName` и `diskURI`, поддерживаются такие опциональные поля:

- ☐ `cachingMode` — режим кэширования диска. Допустимы значения `None`, `ReadOnly` и `ReadWrite`, по умолчанию `None`;
- ☐ `fsType` — тип подключаемой системы, по умолчанию `ext4`;
- ☐ `readOnly` — определяет, доступна ли файловая система только для чтения, по умолчанию `false`.

Максимальный объем дисков в Azure — 1023 Гбайт. Виртуальная машина Azure позволяет подключить до 16 дисков. Каждый диск может использоваться только одной ВМ.

Файловое хранилище Azure

Помимо дисков, Azure предоставляет разделяемую файловую систему, аналогичную AWS EFS. Однако в качестве протокола в ней применяется SMB/CIFS (поддерживаются SMB 2.1 и SMB 3.0). Она основана на платформе хранения Azure и имеет такие же доступность, устойчивость, масштабируемость и поддержку географического распределения, как и Azure Blob, Table или Queue.

Для использования файлового хранилища Azure на каждой клиентской ВМ нужно установить пакет `cifs-utils`. Еще придется создать объект `secret`, так как это обязательный параметр:

```

apiVersion: v1
kind: Secret
metadata:
  name: azure-file-secret
type: Opaque
data:
  azurestorageaccountname: <base64 encoded account name>
  azurestorageaccountkey: <base64 encoded account key>

```


Далее показан конфигурационный файл для файлового хранилища Azure:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - image: some-container
    name: some-container
    volumeMounts:
    - name: some-volume
      mountPath: /azure
  volumes:
  - name: some-volume
    azureFile:
      secretName: azure-file-secret
      shareName: azure-share
      readOnly: false
```

Файловое хранилище Azure поддерживает разделение внутри одного региона и подключение внешних клиентов.

Тема GlusterFS и Ceph в Kubernetes

GlusterFS и Ceph — две распределенные системы для постоянного хранения данных. GlusterFS — это по своей сути сетевая файловая система, а Ceph — объектное хранилище. Оба они предоставляют интерфейсы для работы с блоками, объектами и файловыми системами. Внутри для хранения данных используется `xfs`, а метаданные представлены в виде атрибутов `xattr`. Есть несколько ситуаций, когда в кластере Kubernetes имеет смысл применять тома на основе GlusterFS или Ceph.

- ❑ У вас есть множество данных в GlusterFS или Ceph и приложений, которые их используют.
- ❑ Вы имеете опыт администрирования и работы с GlusterFS или Ceph.
- ❑ Вы задействуете облачную платформу, но вас не устраивают ограничения ее постоянного хранилища.

Использование GlusterFS

В системе GlusterFS сделан акцент на простоту. Она предоставляет каталоги в их исходном виде, оставляя заботу о высокой доступности, репликации и распределении клиентам или различным прослойкам. GlusterFS группирует данные по логическим томам, каждый из них может охватывать несколько узлов (компьютеров) и состоит из блоков, хранящих файлы. Распределение файлов по блокам выполняется согласно DHT (distributed hash table — распределенная хеш-таблица).

При переименовании, а также расширении или изменении балансировки кластера GlusterFS файлы могут перемещаться между блоками. На рис. 7.2 показана структура GlusterFS.

Чтобы применять GlusterFS в качестве постоянного хранилища в Kubernetes, необходимо выполнить несколько шагов (при условии, что у вас уже есть рабочий кластер GlusterFS). При этом узлы GlusterFS управляются в виде сервиса Kubernetes с помощью дополнения (хотя разработчиков приложений это не должно волновать).

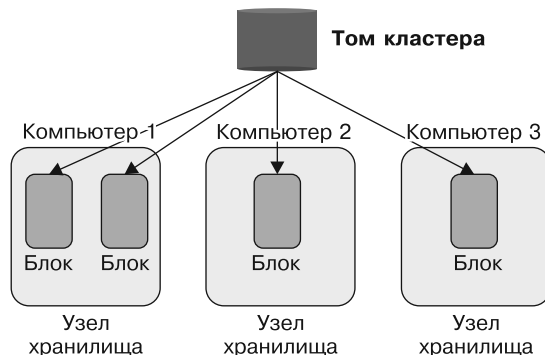


Рис. 7.2

Создание конечных точек

Далее приводится пример конечных точек, которые можно создавать в виде обычных ресурсов Kubernetes с помощью команды `kubectl create`:

```
{
  "kind": "Endpoints",
  "apiVersion": "v1",
  "metadata": {
    "name": "glusterfs-cluster"
  },
  "subsets": [
    {
      "addresses": [
        {
          "ip": "10.240.106.152"
        }
      ],
      "ports": [
        {
          "port": 1
        }
      ]
    }
  ],
}
```

```

    "addresses": [
      {
        "ip": "10.240.79.157"
      }
    ],
    "ports": [
      {
        "port": 1
      }
    ]
  }
}

```

Добавление сервиса для работы с GlusterFS

Чтобы конечные точки были постоянными, ими нужно управлять вручную. Для этого у сервиса Kubernetes не должно быть селектора:

```

{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "glusterfs-cluster"
  },
  "spec": {
    "ports": [
      {"port": 1}
    ]
  }
}

```

Создание подов

Наконец, нужно в разделе `volume` в спецификации пода указать следующую информацию:

```

"volumes": [
  {
    "name": "glusterfsvol",
    "glusterfs": {
      "endpoints": "glusterfs-cluster",
      "path": "kube_vol",
      "readOnly": true
    }
  }
]

```

Контейнеры могут подключать том `glusterfsvol` по имени.

Поле `endpoints` определяет, каким образом дополнение GlusterFS должно искать узлы своего кластера.

Использование Ceph

Для обращения к объектному хранилищу Ceph предусмотрено несколько интерфейсов. Kubernetes поддерживает два из них: *RBD* (блочный интерфейс) и *CEPHFS* (файловая система). На рис. 7.3 показаны разные способы доступа к RADOS (исходному объектному хранилищу). Ceph берет на себя распределение, репликацию и самовосстановление.



Рис. 7.3

Подключение к Ceph с помощью RBD

Kubernetes поддерживает Ceph через интерфейс RBD (RADOS block device — блочное устройство RADOS). На каждом узле вашей системы должен быть установлен пакет `ceph-common`. После запуска кластера Ceph нужно предоставить дополнению Ceph RBD определенную информацию (это делается в конфигурационном файле пода):

- ❑ `monitors` — датчики Ceph;
- ❑ `pool` — имя пула RADOS. Если его не указать, будет использоваться стандартный RBD-пул;
- ❑ `image` — имя образа, созданного интерфейсом RBD;
- ❑ `user` — имя пользователя RADOS, по умолчанию `admin`;
- ❑ `keyring` — путь к файлу `keyring`, по умолчанию `/etc/ceph/keyring`;
- ❑ `secretName` — имя объекта с данными для аутентификации. Если его указать, оно переопределит файл `keyring`. Процесс создания подобных объектов рассмотрим чуть позже;

- ❑ `fsType` — тип файловой системы (`ext4`, `xfs` и т. д.), под которую форматируется устройство;
- ❑ `readOnly` — должна ли файловая система быть доступной только для чтения.

Объект `secret`, который Ceph может использовать для аутентификации, выглядит так:

```
apiVersion: v1
kind: Secret
metadata:
  name: ceph-secret
type: "kubernetes.io/rbd"
data:
  key: QVFCMTZWVZvRjVtRXhBQTVrQ1FzN2JCajhWUxSdzI2Qzg0SEE9PQ==
```



В качестве типа объекта `secret` указан `kubernetes.io/rbd`.

Раздел `volumes` в спецификации пода выглядит следующим образом:

```
"volumes": [
  {
    "name": "rbdpd",
    "rbd": {
      "monitors": [
        "10.16.154.78:6789",
        "10.16.154.82:6789",
        "10.16.154.83:6789"
      ],
      "pool": "kube",
      "image": "foo",
      "user": "admin",
      "secretRef": {
        "name": "ceph-secret"
      },
      "fsType": "ext4",
      "readOnly": true
    }
  }
]
```

Ceph RBD поддерживает режимы доступа `ReadWriteOnce` и `ReadOnlyMany`.

Работа с Ceph с помощью CephFS

Если ваш кластер уже сконфигурирован с поддержкой CephFS, можете легко назначить его любому поду. Кроме того, файловая система CephFS совместима с режимами доступа `ReadWriteMany`.

Конфигурация аналогична Ceph RBD, только без пула, образа и типа файловой системы. Поле `secret` может ссылаться на объект `secret` (предпочтительный вариант) или соответствующий файл:

```
apiVersion: v1
kind: Pod
metadata:
  name: cephfs
spec:
  containers:
    - name: cephfs-rw
      image: kubernetes/pause
      volumeMounts:
        - mountPath: "/mnt/cephfs"
          name: cephfs
  volumes:
    - name: cephfs
      cephfs:
        monitors:
          - 10.16.154.78:6789
          - 10.16.154.82:6789
          - 10.16.154.83:6789
        user: admin
        secretFile: "/etc/ceph/admin.secret"
        readOnly: true
```

В разделе `cephfs` можно также указать параметр `path` — по умолчанию `/`.

Механизм выделения RBD существует в двух экземплярах: как часть платформы Kubernetes и в репозитории `external-storage`, который входит в проект Kubernetes incubator.

Управление томами внутрикластерных контейнеров с помощью Flocker

До сих пор мы обсуждали решения для хранения данных за пределами кластера Kubernetes, за исключением непостоянных хранилищ `emptyDir` и `HostPath`. Немного иной подход предлагает Flocker. Эта система проектировалась с прицелом на Docker. Она позволяет перемещать тома Docker вместе с контейнерами, когда те мигрируют между узлами. При переходе на Kubernetes с других платформ оркестрации, таких как Docker compose или Mesos, можно использовать специальное дополнение, которое позволит управлять хранилищами данных с помощью Flocker. Лично я считаю, что Flocker и Kubernetes заметно дублируют друг друга в отношении абстрактных хранилищ.

Flocker состоит из управляющего сервиса и агентов, размещаемых на каждом узле. Это очень похоже на API-сервер и агенты Kubelet в Kubernetes. Управляющий сервис Flocker предоставляет REST API и отвечает за конфигурацию состояния во всем кластере. Агенты следят за тем, чтобы состояние их узлов соответствовало

конфигурации. Например, если узлу X требуется какой-то набор данных, агент Flocker на этом узле его создаст.

Архитектура Flocker проиллюстрирована на рис. 7.4.

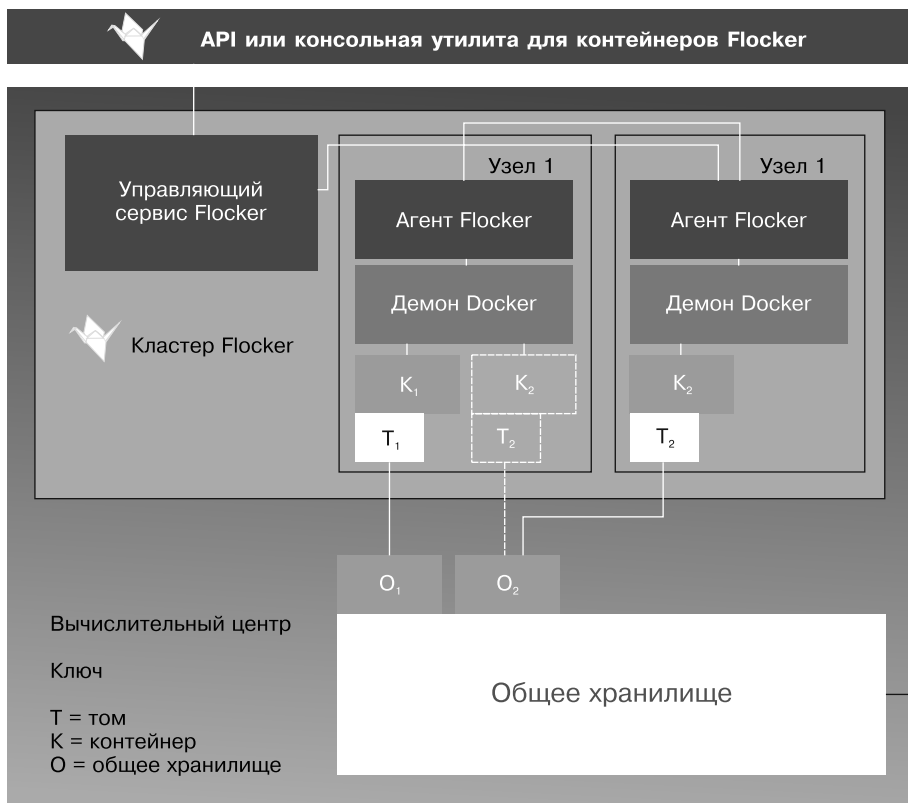


Рис. 7.4

Для управления постоянными томами в Kubernetes понадобится кластер Flocker, который сначала нужно сконфигурировать. Flocker поддерживает множество видов хранилищ (опять же прослеживается сходство со стандартными постоянными томами Kubernetes).

Затем нужно создать наборы данных Flocker, которые можно будет подключать в качестве постоянных томов. По сравнению с тяжелой работой, проделанной ранее, это не составит большого труда. Нужно лишь указать имя для набора данных:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
```

```

- name: some-container
  image: kubernetes/pause
  volumeMounts:
    # name must match the volume name below
    - name: flocker-volume
      mountPath: "/flocker"
volumes:
- name: flocker-volume
  flocker:
    datasetName: some-flocker-dataset

```

Интеграция промышленного хранилища в Kubernetes

Если у вас уже есть *сеть хранения данных* (Storage Area Network, SAN), доступная через интерфейс iSCSI, можете подключить ее к Kubernetes в виде тома, используя дополнение. Здесь применяется та же модель, что и в других дополнениях для разделяемых постоянных хранилищ, рассмотренных ранее. Вы должны сконфигурировать инициатор iSCSI, но для этого не обязательно предоставлять связанную с ним информацию. Достаточно лишь указать следующие параметры:

- ❑ IP-адрес и порт (по умолчанию 3260) iSCSI-интерфейса;
- ❑ *iqn* цели (полное имя участника взаимодействия) — обычно зарезервированное доменное имя;
- ❑ LUN (logical unit number) — номер объекта внутри цели;
- ❑ тип файловой системы;
- ❑ булев флаг *readOnly*.

Дополнение для iSCSI поддерживает режимы *ReadWriteOnce* и *ReadOnlyMany*. Стоит отметить, что создание разделов на устройстве пока не поддерживается. Вот спецификация тома:

```

volumes:
- name: iscsi-volume
  iscsi:
    targetPortal: 10.0.2.34:3260
    iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
    lun: 0
    fsType: ext4
    readOnly: true

```

Отображение томов

Вы можете отобразить несколько томов на один и тот же каталог, представляя их в виде единого тома. Поддерживаются следующие типы томов: *secret*, *downwardAPI* и *configMap*. Это полезно при подключении к поду нескольких источников инфор-

мации. Вместо создания отдельных томов для каждого источника объедините их в один отображенный том, например:

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: busybox
      volumeMounts:
        - name: all-in-one
          mountPath: "/projected-volume"
          readOnly: true
  volumes:
    - name: all-in-one
      projected:
        sources:
          - secret:
              name: the-secret
              items:
                - key: username
                  path: the-group/the-user
          - downwardAPI:
              items:
                - path: "labels"
                  fieldRef:
                    fieldPath: metadata.labels
                - path: "cpu_limit"
                  resourceFieldRef:
                    containerName: the-container
                    resource: limits.cpu
          - configMap:
              name: the-configmap
              items:
                - key: config
                  path: the-group/the-config
```

Использование сторонних хранилищ с помощью FlexVolume

Дополнение FlexVolume стало общедоступным в Kubernetes 1.8. Оно позволяет задействовать сторонние хранилища через унифицированный API. Провайдеры хранилищ предоставляют драйвер, который устанавливается на всех узлах. FlexVolume может динамически обнаруживать имеющиеся драйверы. Далее приведен пример привязки внешнего тома NFS с помощью FlexVolume:

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: nginx-nfs
  namespace: default
spec:
  containers:
  - name: nginx-nfs
    image: nginx
    volumeMounts:
    - name: test
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: test
    flexVolume:
      driver: "k8s/nfs"
      fsType: "nfs"
      options:
        server: "172.16.0.25"
        share: "dws_nas_scratch"

```

Container Storage Interface

Container Storage Interface (CSI) — это попытка стандартизировать взаимодействие между механизмами оркестрации контейнеров и провайдерами хранилищ. Такое решение продвигают команды Kubernetes, Docker, Mesos и Cloud Foundry. Смысл его в том, что провайдеры хранилищ реализуют одно дополнение, а платформы оркестрации контейнеров обязуются поддерживать интерфейс CSI. Это то же самое, что CNI, только для хранилищ. Данный подход имеет несколько преимуществ по сравнению с FlexVolume.

- ❑ CSI — это общепринятый промышленный стандарт.
- ❑ Для развертывания драйверов дополнениям FlexVolume требуется доступ к узлу и ведущей корневой файловой системе.
- ❑ Драйверы FlexVolume часто имеют много внешних зависимостей.
- ❑ FlexVolume предоставляет неуклюжий интерфейс в императивном стиле.

Alpha-версия дополнения CSI впервые появилась в Kubernetes 1.9, а уже в Kubernetes 1.10 перешла в статус бета. Поддержка FlexVolume сохранится для обратной совместимости, по крайней мере на ближайшее будущее. Но по мере того, как интерфейс CSI набирает популярность, все больше провайдеров хранилищ реализуют CSI-драйверы, поэтому я уверен, что все поддерживаемые CSI-дополнения станут частью проекта Kubernetes и доступ к любому хранилищу вскоре можно будет получить через их драйверы.

Представленная на рис. 7.5 схема иллюстрирует принцип работы CSI внутри Kubernetes.

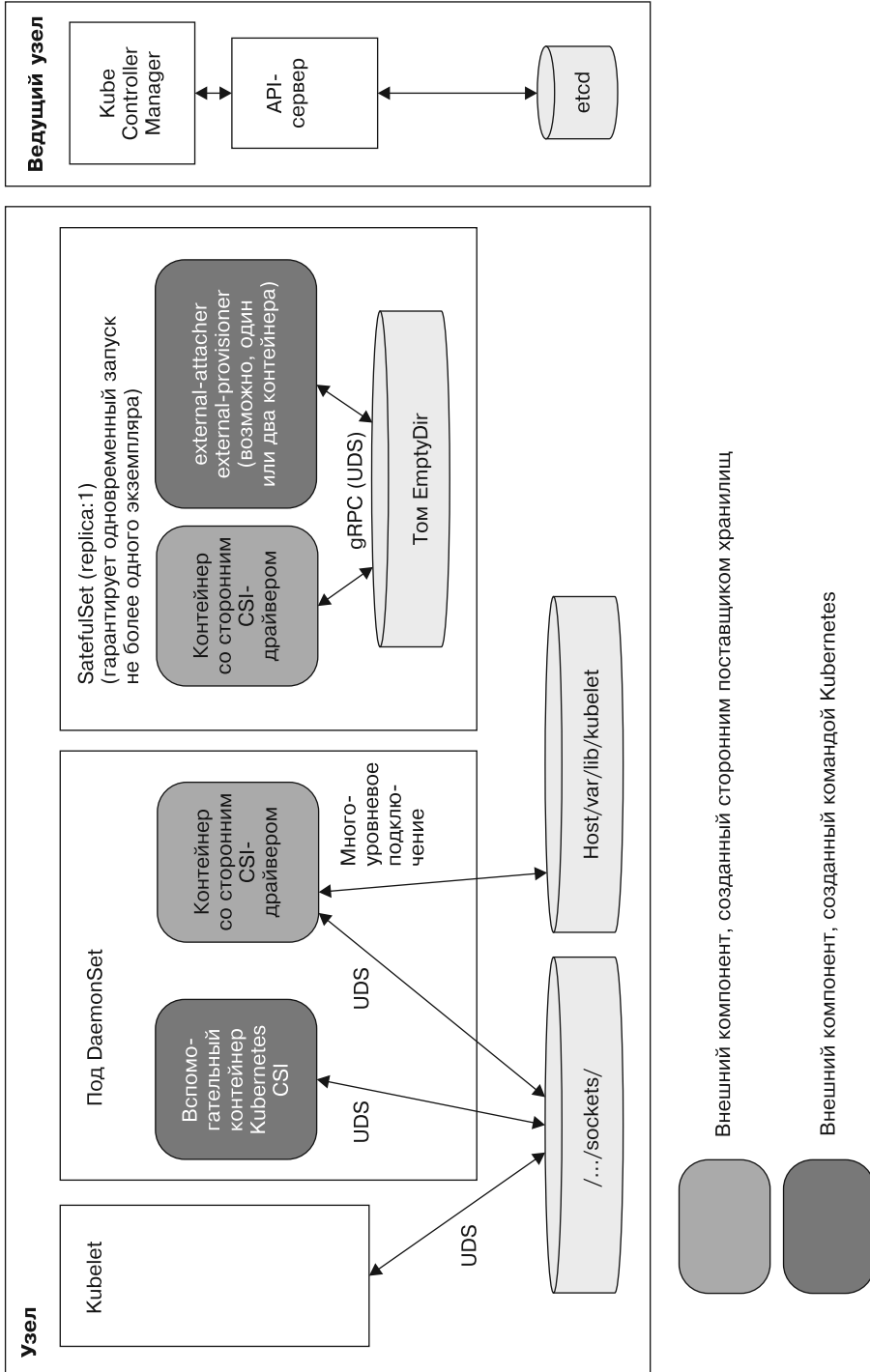


Рис. 7.5

Резюме

В этой главе мы подробно рассмотрели хранение данных в Kubernetes. Вы познакомились с общей абстрактной моделью, основанной на томах, запросах и классах хранилищ, а также с реализацией дополнений для работы с томами. Все хранилища в Kubernetes в итоге подключаются к контейнерам в виде файловой системы или блочного устройства. Такой прямолинейный подход позволяет администраторам конфигурировать и подключать любые системы хранения данных, начиная с локальных каталогов и заканчивая облачными и промышленными хранилищами. Миграция механизмов выделения хранилищ в отдельные проекты положительно влияет на экосистему. Теперь у вас должно сформироваться четкое представление об устройстве и реализации хранилищ в Kubernetes. И вы должны смочь выбрать для своего кластера наиболее подходящее из них.

В главе 8 вы увидите еще один уровень абстракции поверх хранилища, основанный на концепции `StatefulSet`. Он поможет вам в разработке, развертывании и администрировании приложений, которые хранят свое состояние.

8

Запуск приложений с сохранением состояния

Из данной главы вы узнаете, какие меры необходимо предпринять для запуска приложений с сохранением состояния. Kubernetes берет на себя существенную часть работы, автоматически запуская и перезапуская поды в узлах кластера, которые этого требуют, учитывая сложные критерии — пространства имен, лимиты и квоты. Но когда поды выполняют программы, самостоятельно сохраняющие свои данные (например, базы данных и очереди), их перемещение может привести к системному сбою. Прежде всего вы разберетесь в том, что представляют собой поды с сохранением состояния и почему управление ими в Kubernetes оказывается непростым делом. Мы рассмотрим механизмы ограничения этой сложности — общие переменные окружения и DNS-записи. В некоторых ситуациях могут помочь запросы постоянного хранилища, копирования состояния в память и DaemonSet. Основное решение для подов с состоянием, которое продвигает команда Kubernetes, — ресурс StatefulSet (ранее известный как PetSet). Он позволяет управлять индексируемой коллекцией подов с помощью стабильных свойств. В конце вы подробно рассмотрите полноценный пример запуска кластера Cassandra поверх Kubernetes.

Состояние приложений в Kubernetes

Под приложениями без состояния понимают программы, которые не управляют своими данными в рамках Kubernetes. Их состояние находится за пределами кластера и каким-то образом доступно для контейнеров. Из этого раздела вы узнаете, почему управление состоянием играет такую важную роль в проектировании распределенных систем и какие преимущества дает выполнение этого процесса в рамках кластера Kubernetes.

Активная работа с данными в распределенных приложениях

Начнем с основ. Распределенное приложение — это набор процессов, которые выполняются на разных компьютерах, обрабатывают ввод, манипулируют данными, предоставляют API и, возможно, имеют дополнительные функции. Каждый

процесс состоит из программы, среды выполнения и ввода/вывода. Типичные программы получают ввод в виде аргументов командной строки, вероятно, они также считывают файл или обращаются к базе данных. Затем результаты выводятся на экран или записываются в файл или БД. Некоторые программы хранят свое состояние в памяти и способны обслуживать сетевые запросы. В простейшем случае процесс находится на одном компьютере, хранит все свое состояние в памяти или считывает его из файла, а в качестве среды выполнения выступает операционная система. Если такой процесс выйдет из строя, его придется перезапустить вручную. Он привязан к своему компьютеру. Распределенные приложения — совсем другое дело. Одного компьютера недостаточно для своевременной обработки всех данных или обслуживания всех запросов — он может просто не вместить всю нужную информацию. Данные могут быть настолько объемными, что загружать их на каждый узел для дальнейшей обработки будет слишком расточительно. Физические устройства способны ломаться и требовать замены. Компьютеры, занимающиеся обработкой, нужно обновлять. Пользователи зачастую разбросаны по всему миру.

Если учесть все сказанное, становится ясно, что традиционный подход здесь не годится. Данные становятся узким местом. Пользователи/клиенты должны получать лишь краткую сводку или уже обработанную информацию. Все интенсивные вычисления следует выполнять в непосредственной близости к данным, поскольку их перемещение оказывается недопустимо медленным и затратным. Таким образом, основная часть обрабатывающего кода должна выполняться в одном вычислительном центре и в одном сетевом окружении с данными.

Зачем управлять состоянием в Kubernetes

Основная причина необходимости управления состоянием внутри Kubernetes, а не в отдельном кластере состоит в том, что данная платформа предоставляет значительную часть инфраструктуры, необходимой для мониторинга, масштабирования, выделения, защиты и администрирования хранилища. Запуск параллельного кластера приведет к дублированию множества функций.

Зачем выносить управление состоянием за пределы Kubernetes

Не будем исключать другой вариант. В некоторых ситуациях управление состоянием в отдельном кластере, неподконтрольном Kubernetes, может быть оправданным, если этот кластер находится в той же внутренней сети (близость размещения данных перевешивает все остальное).

Некоторые потенциальные сценарии таковы.

- ❑ Отдельный кластер для хранения данных уже существует, и вы не хотите менять то, что и так работает.

- ❑ Кластер с хранилищем используется другими приложениями, не основанными на Kubernetes.
- ❑ Поддержка Kubernetes в хранилище недостаточно стабильная или развитая.

Перенос приложений, хранящих свое состояние в отдельном кластере, в Kubernetes имеет смысл выполнять постепенно, делая интеграцию все более тесной.

Механизм обнаружения: общие переменные окружения или DNS-записи

Kubernetes предоставляет несколько механизмов глобального обнаружения в рамках кластера. Если хранилище находится за пределами Kubernetes, поды должны знать, как его найти и как к нему обращаться. Существует два основных метода:

- ❑ DNS;
- ❑ переменные окружения.

В некоторых случаях их можно сочетать, при этом переменные окружения переопределяют DNS.

Обращение к внешним хранилищам данных через DNS

Метод, основанный на DNS, прост и прямолинеен. Если кластер с хранилищем использует балансировщик нагрузки и предоставляет стабильную конечную точку, экземпляры подов могут обращаться к ней напрямую, подключаясь к внешнему кластеру.

Обращение к внешним хранилищам данных через переменные окружения

Еще один простой подход состоит в передаче информации о подключении к внешнему хранилищу с помощью переменных окружения. Kubernetes предоставляет ресурс `ConfigMap` (карта конфигурации) для хранения конфигурации отдельно от образа контейнера. Конфигурация состоит из пар «ключ — значение» и может быть доступна в контейнерах и томах в виде переменных окружения. Если вы хотите держать в тайне сведения о соединении, лучше использовать объекты `secret`.

Создание ресурса `ConfigMap`. Из представленного далее описания создается конфигурационный файл, хранящий список адресов:

```
apiVersion: v1
kind: ConfigMap
```

```

metadata:
  name: db-config
  namespace: default
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8

> kubectl create -f .\configmap.yaml configmap
"db-config" created

```

Раздел `data` содержит все пары «ключ — значение» (в данном случае это всего одна пара с ключом `db-ip-addresses`). Он пригодится позже, при задании поля `configmap` в поде. Можете проверить содержимое файла, чтобы убедиться в его корректности:

```

> kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8
kind: ConfigMap
metadata:
  creationTimestamp: 2017-01-09T03:14:07Z
  name: db-config
  namespace: default
  resourceVersion: "551258"
  selfLink: /api/v1/namespaces/default/configmaps/db-config
  uid: aebcc007-d619-11e6-91f1-3a7ae2a25c7d

```

`ConfigMap` создается и другими способами, например с помощью аргументов командной строки `--from-value` или `--from-file`.

Использование ConfigMap в виде переменной окружения

При создании пода можно указать ресурс `ConfigMap` и затем работать с ним разными способами. Далее приведен пример подключения карты конфигурации в виде переменной окружения:

```

apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
    - name: some-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: DB_IP_ADDRESSES
          valueFrom:

```



```
configMapKeyRef:
  name: db-config
  key: db-ip-addresses
restartPolicy: Never
```

Этот под с минимальным контейнером `busybox` внутри запускает команду `env` `bash` и сразу прекращает работу. Ключ `db-ip-addresses` из карты `db-config` привязывается к переменной окружения `DB_IP_ADDRESSES`, которую можно видеть в следующем выводе:

```
> kubectl logs some-pod
HUE_REMINDERS_SERVICE_PORT=80
HUE_REMINDERS_PORT=tcp://10.0.0.238:80
KUBERNETES_PORT=tcp://10.0.0.1:443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=some-pod
SHLVL=1
HOME=/root
HUE_REMINDERS_PORT_80_TCP_ADDR=10.0.0.238
HUE_REMINDERS_PORT_80_TCP_PORT=80
HUE_REMINDERS_PORT_80_TCP_PROTO=tcp
DB_IP_ADDRESSES=1.2.3.4,5.6.7.8
HUE_REMINDERS_PORT_80_TCP=tcp://10.0.0.238:80
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.0.0.1:443
HUE_REMINDERS_SERVICE_HOST=10.0.0.238
PWD=/
KUBERNETES_SERVICE_HOST=10.0.0.1
```

Резервная копия состояния в памяти

В некоторых случаях имеет смысл временно сохранять состояние в памяти. Распространенными примерами являются распределенное кэширование и хранение информации, которая зависит от времени. В таких ситуациях потребность в постоянном хранилище отпадает — создание нескольких подов с доступом через сервис способно стать верным решением. Мы можем применить стандартный механизм Kubernetes — метки, чтобы определять поды, хранящие копии одного и того же состояния, и предоставлять к ним доступ с помощью сервиса. Если под выйдет из строя, Kubernetes заменит его новым, а пока он синхронизируется, обслуживанием состояния будут заниматься другие экземпляры пода. Мы даже можем воспользоваться альфа-версией механизма не принадлежности подов, чтобы поды с одинаковым состоянием не развертывались на одном и том же узле.

Использование DaemonSet в качестве резервного постоянного хранилища

Некоторые приложения, такие как распределенные базы данных или очереди, хранят несколько копий своего состояния и автоматически синхронизируют свои узлы (позже мы во всех подробностях рассмотрим Cassandra). В таких случаях важно, чтобы поды развертывались на отдельных узлах. Следует также убедиться в том, что эти поды развертываются на узлах с определенной аппаратной конфигурацией или даже на специально выделенных для этого устройствах. Для таких ситуаций идеально подходит DaemonSet. Мы можем пометить группу узлов так, чтобы каждый из ее участников выполнял лишь один под с состоянием.

Подключение постоянных томов

Если приложение поддерживает эффективное хранение состояния в постоянном хранилище, подключение постоянных томов — подходящее решение (см. главу 7). Приложению, хранящему состояние, будет предоставлен подключенный том, который выглядит как локальная файловая система.

Применение StatefulSet

Контроллер StatefulSet появился в Kubernetes относительно недавно (в версии 1.3 он был представлен как PetSets, а в Kubernetes 1.5 его переименовали в StatefulSet). Он специально предназначен для поддержки приложений с состоянием, в которых подлинность участников имеет значение, и при этом поды должны сохранять свою идентичность в наборе даже после перезапуска. StatefulSet обеспечивает упорядоченное развертывание и масштабирование. В отличие от обычных подов участники StatefulSet привязаны к постоянному хранилищу.

Когда имеет смысл использовать StatefulSet

StatefulSet отлично подходит для приложений, к которым предъявляется хотя бы одно из следующих требований:

- ☐ наличие стабильных уникальных сетевых идентификаторов;
- ☐ наличие стабильного постоянного хранилища;
- ☐ упорядоченное элегантное развертывание и масштабирование;
- ☐ упорядоченное элегантное удаление и прерывание работы.

Компоненты StatefulSet

Для получения рабочего контроллера StatefulSet необходимо правильно сконфигурировать несколько компонентов:

- ❑ лишенный интерфейса сервис, ответственный за сетевую идентификацию подов внутри StatefulSet;
- ❑ сам контроллер StatefulSet с несколькими репликами;
- ❑ механизм выделения постоянного хранилища — автоматический или с участием администратора.

Далее приведен пример сервиса `nginx`, который будет использоваться в StatefulSet:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
```

Теперь укажем ссылку на сервис в конфигурационном файле StatefulSet:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
```

Дальше идет шаблон пода с подключенным томом, который называется `www`:

```
spec:
  terminationGracePeriodSeconds: 10
  containers:
    - name: nginx
      image: gcr.io/google_containers/nginx-slim:0.8
      ports:
        - containerPort: 80
```

```

name: web
volumeMounts:
- name: www
  mountPath: /usr/share/nginx/html

```

Последний, но тем не менее важный раздел `volumeClaimTemplates` использует запрос `www`, который совпадает с подключенным томом. В запросе указаны 1 Гбайт хранилища (`storage`) и доступ на чтение и запись (`ReadWriteOnce`):

```

volumeClaimTemplates:
- metadata:
  name: www
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gib

```

Выполнение кластера Cassandra в Kubernetes

В этом разделе мы подробно рассмотрим масштабный пример подготовки кластера Cassandra для работы с Kubernetes. Проект находится по адресу <https://github.com/kubernetes/kubernetes/tree/master/examples/storage/cassandra>.

Для начала познакомимся с системой Cassandra и ее характерными чертами, а затем пошагово пройдемся по процессу ее настройки и запуска, используя некоторые методики и стратегии, рассмотренные в предыдущем разделе.

Краткое введение в Cassandra

Cassandra — это распределенное столбчатое хранилище, изначально рассчитанное на большие данные. Оно отличается высокой скоростью, надежностью (нет единой точки отказа), доступностью и способностью к линейному масштабированию. А также поддерживает размещение в нескольких вычислительных центрах. Все это достигается за счет кристально четкой специализации, тщательной разработки поддерживаемых функций и, что не менее важно, отсеечения лишних возможностей. В своей предыдущей компании я работал с кластером Kubernetes, который использовал Cassandra в качестве основного хранилища для показаний датчиков (около 100 Тбайт). Cassandra распределяет данные по набору (кольцу) узлов, применяя алгоритм *распределенных хеш-таблиц* (distributed hash table, DHT). Узлы общаются между собой по протоколу `gossip`, чтобы быстро получить сведения об общем состоянии кластера (какие части кластера подключились, отключились или недоступны). Cassandra постоянно уплотняет данные и балансирует кластер. Обычно информация реплицируется для достижения избыточности, устойчивости и высокой доступности. С точки зрения разработчика Cassandra отлично подходит для хранения временных рядов и предоставляет модель, в которой для каждого

запроса можно указывать свой уровень согласованности. Это идемпотентное хранилище (крайне важное свойство для распределенной базы данных), то есть записи можно вставлять и обновлять по несколько раз.

На рис. 8.1 показана структура кластера Cassandra. Обратите внимание на то, как клиенты обращаются к произвольным узлам и как их запросы автоматически направляются к узлу с нужными данными.

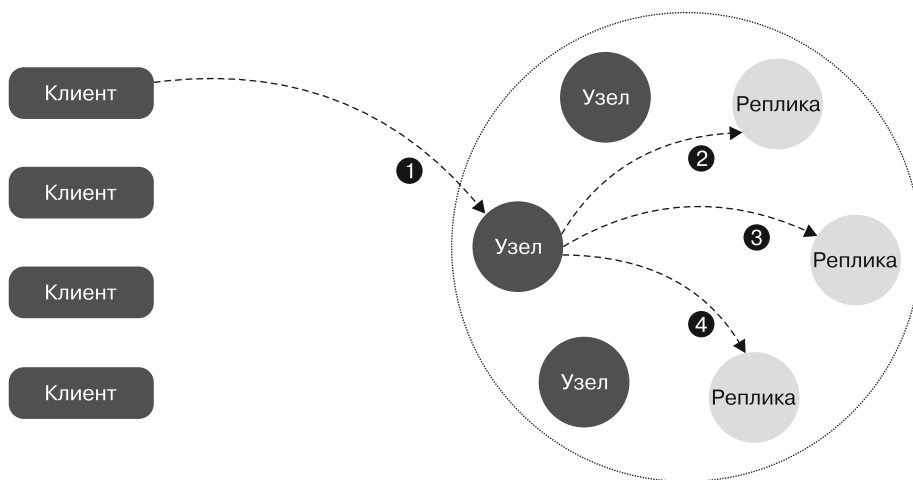


Рис. 8.1

Docker-образ Cassandra

Чтобы развернуть Cassandra в Kubernetes, понадобится специальный образ Docker. Это важный нюанс, поскольку так мы сможем использовать Kubernetes для отслеживания подов Cassandra. Образ доступен по адресу github.com/kubernetes/kubernetes/tree/master/examples/storage/cassandra/image.

Рассмотрим ключевые участки Docker-файла. Образ основан на Ubuntu Slim:

```
FROM gcr.io/google_containers/ubuntu-slim:0.9
```

Добавим и скопируем необходимые файлы (Cassandra.jar, различные конфигурационные файлы, скрипты для запуска и мониторинга), создадим каталог, в котором Cassandra будет хранить свои таблицы (SSTable), и подключим его:

```
ADD files /
```

```
RUN set -e && echo 'debconf debconf/frontend select Noninteractive' |
debconf-set-selections \
    && apt-get update && apt-get -qq -y --force-yes install --no-
installrecommends
\
```

```

openjdk-8-jre-headless \
libjemalloc1 \
localepurge \
wget && \
mirror_url=$( wget -q -O -
http://www.apache.org/dyn/closer.cgi/cassandra/ \
| sed -n 's#.*href="\(\http://.*cassandra\[^\]*\)".*#\1#p' \
| head -n 1 \
) \
&& wget -q -O - ${mirror_url}/${CASSANDRA_VERSION}/apache-cassandra-
${CASSANDRA_VERSION}-bin.tar.gz \
| tar -xzf - -C /usr/local \
&& wget -q -O -
https://github.com/Yelp/dumb-init/releases/download/v${DI_VERSION}/dumb-init_
t_${DI_VERSION}_amd64 > /sbin/dumb-init \
&& echo "$DI_SHA /sbin/dumb-init" | sha256sum -c - \
&& chmod +x /sbin/dumb-init \
&& chmod +x /ready-probe.sh \
&& mkdir -p /cassandra_data/data \
&& mkdir -p /etc/cassandra \
&& mv /logback.xml /cassandra.yaml /jvm.options /etc/cassandra/ \
&& mv /usr/local/apache-cassandra-${CASSANDRA_VERSION}/conf/cassandraenv.
sh /etc/cassandra/ \
&& adduser --disabled-password --no-create-home --gecos '' --disabled-
login cassandra \
&& chown cassandra: /ready-probe.sh \

```

VOLUME ["/\$CASSANDRA_DATA"]

Откроем важные порты для доступа к Cassandra и для того, чтобы узлы могли общаться между собой по протоколу gossip:

```

# 7000: intra-node communication
# 7001: TLS intra-node communication
# 7199: JMX
# 9042: CQL
# 9160: thrift service

```

EXPOSE 7000 7001 7199 9042 9160

В конце выполним следующую команду, основанную на `dumb-init` — простой системе инициализации контейнеров от Yelp. В итоге будет запущен скрипт `run.sh`:

CMD ["/sbin/dumb-init", "/bin/bash", "/run.sh"]

Содержимое скрипта `run.sh`. Скрипт `run.sh` требует некоторых навыков работы с командной оболочкой, но результат его применения оправдывает все усилия. Docker позволяет запускать только одну команду, поэтому, если у вас есть сложные приложения, важно иметь скрипт запуска, который подготовит среду для их выполнения. В данном случае образ поддерживает несколько вариантов

развертывания (StatefulSet, контроллер репликации, DaemonSet), о которых мы поговорим позже, для их выбора в скрипте запуска задействуются переменные окружения.

Для начала укажем некоторые локальные переменные в конфигурационном файле Cassandra /etc/cassandra/cassandra.yaml. Переменная CASSANDRA_CFG будет использоваться в остальной части скрипта:

```
set -e
CASSANDRA_CONF_DIR=/etc/cassandra
CASSANDRA_CFG=$CASSANDRA_CONF_DIR/cassandra.yaml
```

В решениях на основе StatefulSet вместо CASSANDRA_SEEDS определяется переменная HOSTNAME:

```
# we are doing StatefulSet or just setting our seeds
if [ -z "$CASSANDRA_SEEDS" ]; then
    HOSTNAME=$(hostname -f)
fi
```

Затем идет длинный список переменных со значениями по умолчанию в формате \${VAR_NAME}:=<значение>. В первую очередь используется переменная VAR_NAME, если ее нет, берется стандартное значение.

Синтаксис \${VAR_NAME}:=<значение> делает то же самое, но если переменной не существует, создает ее с заданным значением.

Далее применяются оба варианта:

```
CASSANDRA_RPC_ADDRESS="${CASSANDRA_RPC_ADDRESS:-0.0.0.0}"
CASSANDRA_NUM_TOKENS="${CASSANDRA_NUM_TOKENS:-32}"
CASSANDRA_CLUSTER_NAME="${CASSANDRA_CLUSTER_NAME:= 'Test Cluster'}"
CASSANDRA_LISTEN_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_RPC_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_DISK_OPTIMIZATION_STRATEGY="${CASSANDRA_DISK_OPTIMIZATION_STRATEGY:-ssd}"
CASSANDRA_MIGRATION_WAIT="${CASSANDRA_MIGRATION_WAIT:-1}"
CASSANDRA_ENDPOINT_SNITCH="${CASSANDRA_ENDPOINT_SNITCH:-SimpleSnitch}"
CASSANDRA_DC="${CASSANDRA_DC}"
CASSANDRA_RACK="${CASSANDRA_RACK}"
CASSANDRA_RING_DELAY="${CASSANDRA_RING_DELAY:-30000}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:-true}"
CASSANDRA_SEEDS="${CASSANDRA_SEEDS:false}"
CASSANDRA_SEED_PROVIDER="${CASSANDRA_SEED_PROVIDER:-org.apache.cassandra.locator.SimpleSeedProvider}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:false}"

# Отключение авторизации JMX
CASSANDRA_OPEN_JMX="${CASSANDRA_OPEN_JMX:-false}"
# Передача GC на STDOUT
CASSANDRA_GC_STDOUT="${CASSANDRA_GC_STDOUT:-false}"
```

Затем идет раздел, в котором все переменные выводятся на экран. Здесь приведены лишь несколько строчек:

```
echo Starting Cassandra on ${CASSANDRA_LISTEN_ADDRESS}
echo CASSANDRA_CONF_DIR ${CASSANDRA_CONF_DIR}
...
```

Следующий раздел очень важен. По умолчанию Cassandra использует стратегию simple snitch, которая ничего не знает о серверных стойках и вычислительных центрах. Это не самый лучший подход к работе с распределенными кластерами.

Cassandra учитывает физическое размещение узлов вплоть до серверной стойки и может оптимизировать избыточность и высокую доступность. Для этого надлежащим образом ограничивается взаимодействие между вычислительными центрами:

```
# if DC and RACK are set, use GossipingPropertyFileSnitch
if [[ $CASSANDRA_DC && $CASSANDRA_RACK ]]; then
  echo "dc=$CASSANDRA_DC" > $CASSANDRA_CONF_DIR/cassandra-rackdc.properties
  echo "rack=$CASSANDRA_RACK" >> $CASSANDRA_CONF_DIR/cassandra-
    rackdc.properties
  CASSANDRA_ENDPOINT_SNITCH="GossipingPropertyFileSnitch"
fi
```

Важную роль играет управление памятью. Вы можете определить максимальный размер кучи, чтобы хранилище Cassandra не начало сбрасываться на диск:

```
if [ -n "$CASSANDRA_MAX_HEAP" ]; then
  sed -ri "s/^(#)?-Xmx[0-9]+.*?/-Xmx$CASSANDRA_MAX_HEAP/"
    "$CASSANDRA_CONF_DIR/jvm.options"
  sed -ri "s/^(#)?-Xms[0-9]+.*?/-Xms$CASSANDRA_MAX_HEAP/"
    "$CASSANDRA_CONF_DIR/jvm.options"
fi

if [ -n "$CASSANDRA_REPLACE_NODE" ]; then
  echo "-Dcassandra.replace_address=$CASSANDRA_REPLACE_NODE/" >>
    "$CASSANDRA_CONF_DIR/jvm.options"
fi
```

Информация о стойках и вычислительных центрах хранится в простом Java-файле properties:

```
for rackdc in dc rack; do
  var="CASSANDRA_${rackdc^^}"
  val="${!var}"
  if [ "$val" ]; then
    sed -ri "s/^(('$rackdc'=').*?/1 '$val'/' '$CASSANDRA_CONF_DIR/
      cassandrarakdc.properties"
  fi
done
```

Следующий раздел проходится по всем определенным ранее переменным, находит подходящие ключи в конфигурационных файлах Cassandra.yaml и переза-

писывает их. Благодаря этому каждый конфигурационный файл подстраивается на лету непосредственно перед запуском Cassandra:

```
for yml in \
  broadcast_address \
  broadcast_rpc_address \
  cluster_name \
  disk_optimization_strategy \
  endpoint_snitch \
  listen_address \
  num_tokens \
  rpc_address \
  start_rpc \
  key_cache_size_in_mb \
  concurrent_reads \
  concurrent_writes \
  memtable_cleanup_threshold \
  memtable_allocation_type \
  memtable_flush_writers \
  concurrent_compactors \
  compaction_throughput_mb_per_sec \
  counter_cache_size_in_mb \
  internode_compression \
  endpoint_snitch \
  gc_warn_threshold_in_ms \
  listen_interface \
  rpc_interface \
; do
  var="CASSANDRA_${yml^^}"
  val="${!var}"
  if [ "$val" ]; then
    sed -ri 's/^(#)?("$yml":).*\/2 "$val"/' "$CASSANDRA_CFG"
  fi
done

echo "auto_bootstrap: ${CASSANDRA_AUTO_BOOTSTRAP}" >> $CASSANDRA_CFG
```

Следующий раздел посвящен определению провайдера (-ов) распространения в зависимости от способа развертывания (StatefulSet или нет). Далее показан небольшой прием, который позволяет первому поду начать распространять самого себя:

```
# Самораспространение. Это нужно сделать только для первого пода;
# все остальные смогут получить объекты seed от провайдера распространения.
if [[ $CASSANDRA_SEEDS == 'false' ]]; then
  sed -ri 's/- seeds:.*\/- seeds: "'$POD_IP'"/' $CASSANDRA_CFG
else # if we have seeds set them. Probably StatefulSet
  sed -ri 's/- seeds:.*\/- seeds: "'$CASSANDRA_SEEDS'"/' $CASSANDRA_CFG
fi

sed -ri 's/- class_name: SEED_PROVIDER\/- class_name:
'"$CASSANDRA_SEED_PROVIDER'"/' $CASSANDRA_CFG
```

За ним идет раздел, в котором выбираются варианты удаленного управления и JMX-мониторинга. Сложные распределенные системы нуждаются в подходящих механизмах администрирования. Cassandra поддерживает широко распространенный стандарт *JMX* (Java Management Extensions):

```
# Направляем сборщик мусора в stdout
if [[ $CASSANDRA_GC_STDOUT == 'true' ]]; then
    sed -ri 's/ -Xloggc:\var\log\cassandra\gc\.log/'
    $CASSANDRA_CONF_DIR/cassandra-env.sh
fi

# Позволяем RMI и JMX работать на одном порте
echo "JVM_OPTS=\"$JVM_OPTS -Djava.rmi.server.hostname=$POD_IP\"" >>
    $CASSANDRA_CONF_DIR/cassandra-env.sh

# Получаем предупреждения с помощью Migration Service
echo "-
Dcassandra.migration_task_wait_in_seconds=${CASSANDRA_MIGRATION_WAIT}" >>
    $CASSANDRA_CONF_DIR/jvm.options
echo "-Dcassandra.ring_delay_ms=${CASSANDRA_RING_DELAY}" >>
    $CASSANDRA_CONF_DIR/jvm.options

if [[ $CASSANDRA_OPEN_JMX == 'true' ]]; then
    export LOCAL_JMX=no
    sed -ri 's/ -Dcom.sun.management.jmxremote.authenticate=true/ -
        Dcom.sun.management.jmxremote.authenticate=false/'
        $CASSANDRA_CONF_DIR/cassandra-env.sh
    sed -ri 's/ -
        Dcom.sun.management.jmxremote.password.file=\etc\cassandra\jmxremot
        e\password/' $CASSANDRA_CONF_DIR/cassandra-env.sh
fi
```

Наконец, укажем JAR-файл Cassandra в CLASSPATH и запустим его в интерактивном режиме (не в виде демона) от имени пользователя Cassandra:

```
export CLASSPATH=/kubernetes-cassandra.jar
su cassandra -c "$CASSANDRA_HOME/bin/cassandra -f"
```

Интеграция Kubernetes и Cassandra

Интеграция систем Kubernetes и Cassandra требует определенных усилий, поскольку последняя специально разрабатывалась для того, чтобы быть максимально автономной. Мы хотим, чтобы кластер Kubernetes смог подключаться в нужный момент и предоставлять возможность автоматического перезапуска неисправных узлов, мониторинга, выделения подов с Cassandra внутри и их унификации с другими подами. Cassandra — сложная платформа с разветвленными средствами управления. Она поставляется вместе с конфигурационным файлом *Cassandra.yaml*, все параметры которого переопределяются с помощью переменных окружения.

Подробно о конфигурации Cassandra

Два параметра, которые нас интересуют больше всего, — это провайдер распространения и стратегия `snitch`. Провайдер распространения отвечает за публикацию списка IP-адресов узлов в кластере. Каждый новый узел подключается к одному из провайдеров (обычно их не меньше трех) и обменивается с ним информацией об остальных узлах в кластере. Эти сведения постоянно обновляются и распространяются между узлами по протоколу `gossip`.

По умолчанию провайдер распространения указывается в `Cassandra.yaml` в виде обычного статического списка IP-адресов. В данном случае это всего лишь локальный сетевой интерфейс:

```
seed_provider:
  - class_name: SEED_PROVIDER
    parameters:
      # Список адресов узлов разделяется запятыми,
      # например: "<ip1>,<ip2>,<ip3>"
      - seeds: "127.0.0.1"
```

Еще один важный параметр — стратегия `snitch`. Она имеет две функции:

- ❑ предоставляет Cassandra сведения о топологии сети, достаточные для эффективной маршрутизации запросов;
- ❑ позволяет Cassandra распространять реплики по кластеру, чтобы избежать цепных сбоев. Для этого узлы группируются по серверным стойкам (не обязательно физическим) и вычислительным центрам. Cassandra делает все для того, чтобы на одной и той же стойке было не больше одной реплики.

Несколько стратегий `snitch` входят в стандартную поставку Cassandra, однако ни одна из них не умеет работать с Kubernetes. По умолчанию указан параметр `SimpleSnitch`, но его можно переопределить:

```
# Вы можете использовать нестандартную стратегию, указав полное имя класса
# snitch (должен находиться в вашей собственной переменной classpath).
endpoint_snitch: SimpleSnitch
```

Собственный провайдер распространения

Когда Cassandra запускается внутри Kubernetes, ее узлы, в том числе узлы распространения, выполняются в виде подов и могут перемещаться. Для этого провайдер распространения должен взаимодействовать с API-сервером Kubernetes.

Далее приводится небольшой фрагмент Java-класса `KubernetesSeedProvider`, который реализует API `SeedProvider`, принадлежащий Cassandra:

```
public class KubernetesSeedProvider implements SeedProvider {
    ...
    /**
     * Вызываем Kubernetes API,
     * чтобы составить список провайдеров распространения
     */
}
```

```

public List<InetAddress> getSeeds() {
    String host = getEnvOrDefault("KUBERNETES_PORT_443_TCP_ADDR",
        "kubernetes.default.svc.cluster.local");
    String port = getEnvOrDefault("KUBERNETES_PORT_443_TCP_PORT", "443");
    String serviceName = getEnvOrDefault("CASSANDRA_SERVICE",
        "cassandra");
    String podNamespace = getEnvOrDefault("POD_NAMESPACE", "default");
    String path = String.format("/api/v1/namespaces/ %s/endpoints/",
        podNamespace);
    String seedSizeVar = getEnvOrDefault("CASSANDRA_SERVICE_NUM_SEEDS", "8");
    Integer seedSize = Integer.valueOf(seedSizeVar);
    String accountToken = getEnvOrDefault("K8S_ACCOUNT_TOKEN",
        "/var/run/secrets/kubernetes.io/serviceaccount/token");
    List<InetAddress> seeds = new ArrayList<InetAddress>();
    try {
        String token = getServiceAccountToken(accountToken);

        SSLContext ctx = SSLContext.getInstance("SSL");
        ctx.init(null, trustAll, new SecureRandom());

        String PROTO = "https://";
        URL url = new URL(PROTO + host + ":" + port + path + serviceName);
        logger.info("Getting endpoints from " + url);
        HttpURLConnection conn = (HttpURLConnection)url.openConnection();

        conn.setSSLSocketFactory(ctx.getSocketFactory());
        conn.setRequestProperty("Authorization", "Bearer " + token);
        ObjectMapper mapper = new ObjectMapper();
        Endpoints endpoints = mapper.readValue(conn.getInputStream(),
            Endpoints.class); }
        ...
    }
    ...
    return Collections.unmodifiableList(seeds);
}

```

Создание неуправляемого сервиса Cassandra

Неуправляемый сервис позволяет клиентам в кластере Kubernetes подключаться к Cassandra через стандартный механизм Kubernetes, который не требует отслеживать подлинность узлов в сети и размещать перед ними балансировщик нагрузки. Kubernetes делает все это автоматически с помощью своих сервисов.

Далее показан конфигурационный файл:

```

apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
    name: cassandra
spec:

```

```
clusterIP: None
ports:
  - port: 9042
selector:
  app: Cassandra
```

Селектор `app: Cassandra` делает участниками сервиса все поды. Kubernetes создаст записи для конечных точек, а DNS вернет запись, доступную для обнаружения. Поле `clusterIP` равно `None`, это означает, что сервис будет неуправляемым и Kubernetes не станет заниматься балансировкой нагрузки или проксированием. Это важно, поскольку в Cassandra узлы взаимодействуют между собой напрямую.

Порт `9042` задействуется в Cassandra для обслуживания CQL-запросов, таких как поиск, добавление/обновление (так называемая операция *upsert*, которая либо добавляет новую запись, либо обновляет уже существующую) и удаление.

Использование контроллера StatefulSet для создания кластера Cassandra

Объявление контроллера StatefulSet — нетривиальная задача. Это, вероятно, самый сложный ресурс в Kubernetes. Он состоит из множества частей: стандартных метаданных, спецификации контроллера, шаблона пода (который часто и сам довольно сложен) и шаблонов запрашивания томов.

Содержимое конфигурационного файла StatefulSet. Шаг за шагом рассмотрим пример конфигурационного файла StatefulSet, который объявляет кластер Cassandra, состоящий из трех узлов.

Сначала идут стандартные метаданные. Обратите внимание на то, что поле `apiVersion` равно `apps/v1` (в версии Kubernetes 1.9 контроллер StatefulSet стал общедоступным):

```
apiVersion: "apps/v1"
kind: StatefulSet
metadata:
  name: cassandra
```

В спецификации StatefulSet определяются имя неуправляемого сервиса, количество подов (поле `replicas`), которые в него входят, и шаблон пода (о нем чуть позже):

```
spec:
  serviceName: cassandra
  replicas: 3
  template: ...
```

Термин `replicas` не очень подходит для подов, поскольку они не являются репликами по отношению друг к другу. Они имеют общий шаблон, но в целом отвечают за разные участки состояния и имеют разные идентификаторы. Путаницу усугубляет тот факт, что в Cassandra `replicas` называют группы узлов, которые дублируют определенный участок состояния, но не являются идентичными,

поскольку каждый из них может управлять дополнительным состоянием. Я начал на GitHub обсуждение с предложением изменить название этого поля с `replicas` на `members`: github.com/kubernetes/website/issues/2103.

Шаблон пода содержит один контейнер, основанный на модифицированном образе Cassandra. Вот как он выглядит:

```
template:
  metadata:
    labels:
      app: cassandra
  spec:
    containers: ...
```

Спецификация контейнера состоит из нескольких важных частей. Вначале идут поля `name` и `image`, которые мы рассматривали ранее:

```
containers:
- name: cassandra
  image: gcr.io/google-samples/cassandra:v12
  imagePullPolicy: Always
```

Дальше определяются разные порты, которые используются узлами Cassandra для внешних и внутренних коммуникаций:

```
ports:
- containerPort: 7000
  name: intra-node
- containerPort: 7001
  name: tls-intra-node
- containerPort: 7199
  name: jmx
- containerPort: 9042
  name: cql
```

В разделе `resources` определяются параметры процессора и памяти, необходимые контейнеру. Это очень важно, потому что подсистема управления хранилищем не должна становиться узким местом из-за нехватки вычислительных ресурсов:

```
resources:
  limits:
    cpu: "500m"
    memory: 1Gi
  requests:
    cpu: "500m"
    memory: 1Gi
```

Cassandra требует доступа к IPC, который запрашивается контейнером через поле `capabilities` в контексте безопасности:

```
securityContext:
  capabilities:
    add:
      - IPC_LOCK
```

В разделе `env` описываются переменные окружения, доступные внутри контейнера. Далее приводится частичный список тех из них, которые являются обязательными. Переменной `CASSANDRA_SEEDS` присваивается имя неуправляемого сервиса, чтобы при запуске узел Cassandra смог подключиться к провайдеру обнаружения и получить сведения обо всем кластере. Заметьте, что в данной конфигурации не используется специальный провайдер обнаружения Kubernetes. Стоит обратить внимание и на переменную `POD_IP`, значение которой присваивается через ссылку в поле `status.podIP` с применением Downward API:

```
env:
- name: MAX_HEAP_SIZE
  value: 512M
- name: CASSANDRA_SEEDS
  value: "cassandra-0.cassandra.default.svc.cluster.local"
- name: POD_IP
  valueFrom:
    fieldRef:
      fieldPath: status.podIP
```

Контейнер выполняет также проверку готовности, чтобы узел Cassandra не получал никаких запросов, пока полностью не запустится:

```
readinessProbe:
  exec:
    command:
    - /bin/bash
    - -c
    - /ready-probe.sh
  initialDelaySeconds: 15
  timeoutSeconds: 5
```

Естественно, кластеру Cassandra нужно читать и записывать данные. Том `cassandra-data` подключается в соответствии со своим местоположением:

```
volumeMounts:
- name: cassandra-data
  mountPath: /cassandra_data
```

На этом спецификация контейнера завершена. Остался только запрос на подключение тома. В данном случае применяется динамическое выделение. Для хранилища Cassandra, особенно для журнала, настоятельно рекомендуется использовать SSD-диски. В нашем примере запрашивается хранилище размером 1 Gi. Методом проб и ошибок я пришел к выводу, что идеальный размер узла Cassandra — 1–2 Тбайт. Дело в том, что в Cassandra происходит активное уплотнение данных с повторной балансировкой. При изменении конфигурации кластера приходится ждать, пока данные не сбалансированы надлежащим образом: отключенному узлу нужно распределить свое содержимое, а новый узел должен быть наполнен. Для этого кластеру Cassandra требуется много места на диске. Рекомендуется оставлять свободным около 50 % дискового пространства. А с учетом репликации (обычно $\times 3$) вместимость хранилища способна превышать объем данных в шесть раз.

В зависимости от вашего авантюризма и предъявляемых требований свободное место можно ограничить 30 % и обойтись двойной репликацией. Но его не должно быть меньше 10 % даже на отдельно взятом узле. Как я узнал из собственного опыта, в этом случае Cassandra просто застопорится и без принятия крайних мер ее узлы не смогут уплотнить и повторно сбалансировать данные.

В качестве режима доступа, естественно, указан `ReadWriteOnce`:

```
volumeClaimTemplates:
- metadata:
  name: cassandra-data
  annotations:
    volume.beta.kubernetes.io/storage-class: fast
spec:
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 1Gi
```

При разворачивании `StatefulSet` в Kubernetes порядок создания подов определяется их порядковыми номерами (индексами). Тот же принцип действует при масштабировании. С точки зрения Cassandra это неважно, поскольку в данной системе узлы подключаются и отключаются в произвольном порядке. Когда в Cassandra удаляется под, его постоянный том сохраняется. Если позже будет создан другой под с тем же индексом, к нему будет подключен тот же постоянный том. Эта устойчивая связь между конкретными экземплярами подов и их хранилищами позволяет Cassandra как следует управлять состоянием.

Распределение Cassandra с помощью контроллера репликации

`StatefulSet` — отличное решение, но, как уже упоминалось, Cassandra — это сложная распределенная база данных. Она имеет множество механизмов для автоматического распределения, балансировки и репликации данных по кластеру. Эти механизмы не оптимизированы для работы с сетевыми хранилищами. Платформа Cassandra проектировалась с расчетом на то, что данные будут храниться непосредственно на узлах. Если узел выходит из строя, кластер может восстановиться за счет резервных данных на других узлах. Рассмотрим иной способ разворачивания Cassandra в Kubernetes, который лучше соответствует семантике этой БД и не требует обновлять Kubernetes до самой свежей версии, чтобы получить поддержку `StatefulSet`.

Мы по-прежнему будем использовать неуправляемый сервис, но на смену `StatefulSet` придет обычный контроллер репликации. Среди важных отличий необходимо отметить следующие.

- ❑ Вместо `StatefulSet` берется контроллер репликации.
- ❑ Хранилище находится на узле, на котором разворачивается под.
- ❑ Используется нестандартный провайдер распространения Kubernetes.

Конфигурационный файл контроллера репликации

Метаданные довольно лаконичные и состоят лишь из имени (метки не требуются):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: cassandra
  # По умолчанию метки
  # берутся из шаблона пода
  # labels:
  # app: Cassandra
```

В разделе `spec` указано поле `replicas`:

```
spec:
  replicas: 3
  # По умолчанию селектор основывается
  # на метках, указанных в шаблоне пода.
  # selector:
  # app: Cassandra
```

Метаданные шаблона пода — это то место, где указывается метка `app: Cassandra`. Контроллер репликации будет следить за тем, чтобы подов с этой меткой было ровно три:

```
template:
  metadata:
    labels:
      app: Cassandra
```

Раздел `spec` шаблона пода описывает список контейнеров. В данном случае указан лишь один контейнер. Он использует тот же Docker-образ Cassandra под названием `cassandra` и выполняет скрипт `run.sh`:

```
spec:
  containers:
    - command:
      - /run.sh
      image: gcr.io/google-samples/cassandra:v11
      name: cassandra
```

В разделе `resources` указано лишь 0,5 вычислительной единицы:

```
resources:
  limits:
    cpu: 0.5
```

Раздел окружения немного отличается. Переменная `CASSANDRA_SEED_PROVIDER` определяет нестандартный класс провайдера распространения Kubernetes, который мы рассматривали ранее. Была также добавлена переменная `POD_NAMESPACE`, извлекающая значение из метаданных с помощью Downward API:

```
env:
  - name: MAX_HEAP_SIZE
```

```

    value: 512M
  - name: HEAP_NEWSIZE
    value: 100M
  - name: CASSANDRA_SEED_PROVIDER
    value: "io.k8s.cassandra.KubernetesSeedProvider"
  - name: POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
  - name: POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP

```

Раздел `ports` оставлен без изменений. Порты `7000` и `7001` выделены для внутреннего взаимодействия, а порт `7199` позволяет подключаться к кластеру Cassandra внешним инструментам, таким как Cassandra OpsCenter. Не забудем также о CQL-порте `9042`, через который клиенты взаимодействуют с кластером:

```

ports:
  - containerPort: 7000
    name: intra-node
  - containerPort: 7001
    name: tls-intra-node
  - containerPort: 7199
    name: jmx
  - containerPort: 9042
    name: cql

```

Том снова подключается к каталогу `/cassandra_data`. Это важно, так как мы используем тот же образ Cassandra, который ожидает, что доступ к каталогу `data` можно получить по определенному пути. С точки зрения Cassandra тип исходного хранилища не имеет значения (хотя вас, как администратора кластера, это должно заботить), так как данные считываются и записываются через интерфейс файловой системы:

```

volumeMounts:
  - mountPath: /cassandra_data
    name: data

```

Главное отличие от решения на основе `StatefulSet` расположено в разделе `volumes`. Раньше мы запрашивали постоянное хранилище, чтобы привязать его к определенному поду с неизменным идентификатором. Вместо этого на узле, на котором происходит разворачивание, контроллер репликации использует `emptyDir`:

```

volumes:
  - name: data
    emptyDir: {}

```

Это имеет далеко идущие последствия. Для каждого узла должно быть выделено достаточно дискового пространства. Если под Cassandra завершит работу, вместе с ним исчезнет и ее хранилище. Даже если перезапустить его на том же физическом или виртуальном компьютере, данные на диске будут потеряны, так как том `emptyDir` удаляется вместе с подом. Стоит отметить, что `emptyDir` сохраняется при перезапуске контейнеров. Так что же происходит после удаления пода? Контроллер репликации запускает новый экземпляр пода с пустым хранилищем. Cassandra обнаружит добавление в кластер нового узла, выделит ему определенную порцию данных и инициирует автоматическую перебалансировку, чтобы переместить эти данные с других узлов. Это ключевое преимущество платформы Cassandra. Она постоянно уплотняет, балансирует и равномерно распределяет информацию по кластеру. Она сама поймет, что нужно сделать, и выполнит все за вас.

Привязка подов к узлам

Основная проблема использования контроллера репликации — то, что несколько подов могут быть развернуты на одном и том же узле Kubernetes. Что произойдет, если степень репликации равна 3 и все три пода, ответственных за определенное пространство ключей, окажутся на одном компьютере? Прежде всего все запросы на чтение и запись в этом диапазоне ключей будут направлены к одному и тому же узлу, создавая повышенную нагрузку. Но хуже всего то, что теряется избыточность и возникает *единая точка отказа*. Если этот узел выйдет из строя, контроллер репликации с радостью запустит три новых пода в каком-то другом месте, но все они будут пустыми и ни один прочий узел в кластере Cassandra не будет иметь подходящих данных, которые можно было бы на них скопировать.

Эта проблема решается с помощью реализуемой в Kubernetes концепции планирования «*непринадлежность*» (*anti-affinity*). Под, который привязывается к узлу, можно пометить таким образом, чтобы он не попал на узел, содержащий экземпляр пода с определенным набором меток. Добавим следующий код в раздел `spec`, чтобы к любому узлу можно было привязать не больше одного пода Cassandra:

```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - cassandra
          topologyKey: kubernetes.io/hostname
```

Использование DaemonSet для распределения Cassandra

Лучшее решение для распределения подов Cassandra по разным узлам — применение контроллера DaemonSet. Как и контроллер репликации, он имеет шаблон пода. Но также содержит селектор, который определяет, на каких узлах следует развертывать его поды. DaemonSet размещает под на каждом узле, который соответствует его селектору, но не позволяет указать определенное количество реплик. Самый простой случай — развертывание экземпляра пода на каждом узле в кластере Kubernetes. Вместе с тем селектор может выбирать подмножество узлов по определенным меткам. Создадим DaemonSet для развертывания Cassandra на кластере Kubernetes:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: cassandra-daemonset
```

Спецификация DaemonSet содержит обычный шаблон пода. Вся магия происходит в разделе `nodeSelector` — в нем мы укажем, что на каждом узле с меткой `app: Cassandra` должен быть развернут ровно один под:

```
spec:
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      # Выбираем только узлы с меткой "app: cassandra":
      nodeSelector:
        app: cassandra
      containers:
```

Все остальное идентично шаблону контроллера репликации. Стоит отметить, что на смену `nodeSelector` вскоре должен прийти раздел `affinity`. Но пока неясно, когда именно это произойдет.

Резюме

Данная глава была посвящена приложениям, сохраняющим свое состояние, и их интеграции с Kubernetes. Вы узнали, что применение этих приложений вызывает дополнительные сложности, и познакомились с механизмами обнаружения — DNS и переменными окружения. Мы также обсудили несколько решений для управления состоянием на основе разных типов хранилищ: постоянного и избыточного, размещенного в памяти. Основная часть главы была связана с развертыванием

кластера Cassandra внутри Kubernetes с использованием механизмов StatefulSet, контроллера репликации и DaemonSet. Каждый из перечисленных подходов имеет свои плюсы и минусы. На этом этапе у вас должно сложиться четкое представление о приложениях, хранящих состояние, и об их применении в системах на основе Kubernetes. В вашем арсенале имеется несколько методик для разных сценариев, и, возможно, вы узнали что-то полезное о Cassandra.

В следующей главе продолжим наше путешествие и исследуем важную тему — масштабирование. В частности, нас будут интересовать автоматическое масштабирование и процесс развертывания/обновления на лету в условиях динамически расширяющегося кластера. Это очень непростые вопросы, особенно когда в кластере выполняются приложения, хранящие свое состояние.

9

Плавающие обновления, масштабирование и квоты

В этой главе мы исследуем автоматизированное масштабирование подов, которое позволяет выполнить Kubernetes, его влияние на плавающие обновления и работу с квотами. Затронем важную тему — выделение ресурсов и управление размером кластера. В конце вы узнаете, как команда разработчиков Kubernetes тестирует свое детище на кластере из 5000 узлов. Далее перечислены основные темы, которые мы будем рассматривать.

- ❑ Горизонтальное автомасштабирование подов.
- ❑ Плавающие обновления с автомасштабированием.
- ❑ Управление дефицитными ресурсами с помощью квот и лимитов.
- ❑ Производительность Kubernetes в экстремальных условиях.

По прочтении этой главы вы сможете проектировать громадные кластеры, экономно выделять для них ресурсы и принимать взвешенные решения по поводу различных компромиссов между производительностью, денежными затратами и доступностью. Вы также получите навыки настройки горизонтального автомасштабирования подов и разумного применения квот на ресурсы. Это позволит кластеру автоматически адаптироваться к непредсказуемым колебаниям нагрузки.

Горизонтальное автомасштабирование подов

Kubernetes способен следить за подами и масштабировать их при достижении предельных показателей использования процессора или каких-то других критериев. Сначала описывается конфигурация автомасштабирования (относительная загруженность процессора и частота проверок), а затем соответствующий контроллер при необходимости регулирует количество реплик.

Рисунок 9.1 иллюстрирует разные элементы автомасштабирования и их отношения.

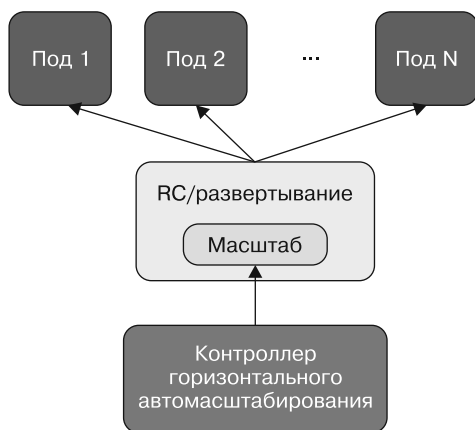


Рис. 9.1

Как видите, контроллер автомасштабирования не занимается непосредственным созданием или удалением подов. В этом он полагается на контроллер репликации или ресурсы развертывания. Это очень элегантное решение, оно исключает ситуацию, когда контроллер репликации и ресурсы развертывания пытаются регулировать количество подов без учета действий, предпринимаемых контроллером автомасштабирования.

Автомасштабирование автоматизирует рутинные операции. Представьте, что контроллер репликации имеет поле `replicas` со значением 3, но средняя нагрузка на процессор требует наличия четырех реплик. В связи с этим пришлось бы вручную обновить шаблон контроллера репликации и продолжить отслеживать загруженность процессора для всех подов. Автомасштабирование делает это за нас.

Объявление горизонтального автомасштабирования подов

Для объявления горизонтального автомасштабирования подов понадобятся контроллер репликации (или ресурс развертывания) и ресурс автомасштабирования. Далее приведен пример конфигурации простого контроллера репликации, который управляет тремя подами с NGINX:

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  template:
    metadata:

```

```

labels:
  run: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80

```

Файл `autoscaling` ссылается на контроллер репликации NGINX с помощью раздела `scaleTargetRef`:

```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  maxReplicas: 4
  minReplicas: 2
  targetCPUUtilizationPercentage: 90
  scaleTargetRef:
    apiVersion: v1
    kind: ReplicationController
    name: nginx

```

Поля `minReplicas` и `maxReplicas` определяют диапазон масштабирования. Это позволяет избегать ситуаций, когда в результате какой-то проблемы количество реплик выходит из-под контроля. Представьте, что в результате ошибки каждый под мгновенно занимает 100 % процессорного времени вне зависимости от реальной нагрузки. Если не указать `maxReplicas`, Kubernetes начнет создавать все новые и новые экземпляры подов, пока не исчерпает ресурсы кластера. Если мы находимся в облачной среде с автомасштабированием виртуальных машин, это способно существенно увеличить наши денежные затраты. У данной проблемы есть и обратная сторона: если не указать `minReplicas`, в периоды затишья все поды будут удаляться, а затем развертываться заново при поступлении новых запросов. Если периоды затишья регулярно чередуются со всплесками активности, цикл будет постоянно повторяться. Это явление можно смягчить, поддерживая минимальное количество запущенных реплик. В предыдущем примере поля `minReplicas` и `maxReplicas` равны 2 и 4 соответственно. Kubernetes будет следить за тем, чтобы количество экземпляров NGINX оставалось в указанных рамках.

У поля `targetCPUUtilizationPercentage` слишком длинное название. Сократим его до `TCUP`. Здесь нужно указать одно число, например 80 %. Если средняя нагрузка соответствует `TCUP`, это способно вызвать постоянные колебания. Kubernetes часто переключается между добавлением и удалением реплик, что нежелательно. Чтобы нивелировать данную проблему, можно указать задержку для увеличения или уменьшения масштаба. Для этого в `kube-controller-manager` предусмотрены два флага:

- ❑ `--horizontal-pod-autoscaler-downscale-delay`. Определяет, как долго контроллер масштабирования должен ждать, прежде чем выполнить следующую операцию уменьшения масштаба. Отсчет начинается с завершения текущей операции. Значение по умолчанию — 5 мин (5m0s);
- ❑ `--horizontal-pod-autoscaler-upscale-delay`. Определяет, сколько контроллер масштабирования должен ждать, прежде чем выполнить следующую операцию увеличения масштаба. Отсчет начинается с завершения текущей операции. Значение по умолчанию — 3 мин (3m0s).

Нестандартные показатели

Использование процессора — это важный показатель для оценки загруженности подов. Если запросов слишком много, размер кластера нужно увеличить, если же поды в основном простаивают, их количество можно урезать. Но это не единственный и часто даже не основной критерий, на который стоит обращать внимание. В качестве ограничения могут выступать память или специфические показатели — глубина внутренней очереди подов на диске, средняя задержка запросов или среднее количество отказов в обслуживании из-за истечения выделенного времени.

Нестандартные показатели для горизонтального масштабирования появились в версии 1.2 как экспериментальное дополнение. В версии 1.6 они достигли состояния бета. Теперь с их помощью можно автоматически масштабировать поды. Контроллер производит масштабирование с учетом всех показателей и в зависимости от максимального количества необходимых реплик.

Использование нестандартных показателей. Горизонтальное автомасштабирование с нестандартными показателями требует дополнительной конфигурации перед запуском кластера. Вначале нужно включить слой агрегации API, затем зарегистрировать API для мониторинга ресурсов и ваших нестандартных показателей. Интерфейс для показателей ресурсов реализован в Heapster — просто запустите эту систему мониторинга с флагом `--api-server=true`. Вам также понадобится отдельный сервер для предоставления интерфейса к нестандартным показателям. Можете начать с этого проекта: <https://github.com/kubernetes-incubator/custom-metrics-apiserver>.

Следующим шагом будет запуск `kube-controller-manager` с такими флагами:

```
--horizontal-pod-autoscaler-use-rest-clients=true
--kubeconfig <путь-к-kubeconfig> OR --master <ip-адрес-api-сервера>
```

Если указать оба флага, у `--master` будет приоритет перед `--kubeconfig`. Эти флаги определяют местоположение слоя агрегации API, позволяя диспетчеру контроллеров общаться с API-сервером.

В Kubernetes 1.7 стандартный слой агрегации работает в одном процессе с `kube-apiserver`, поэтому целевой IP-адрес можно найти следующим образом:

```
> kubectl get pods --selector k8s-app=kube-apiserver --namespace kube-system
-o jsonpath='{.items[0].status.podIP}'
```

Автомасштабирование с помощью kubectl

Утилита `kubectl` способна создать ресурс автомасштабирования с помощью стандартной команды `create` и конфигурационного файла. Но `kubectl` поддерживает также специальную команду `autoscale`, которая с легкостью позволяет задать автомасштабирование без использования специальной конфигурации.

1. Сначала запустим контроллер репликации, который обеспечивает выполнение трех простых реплик с бесконечным циклом `bash-loop`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: bash-loop-rc
spec:
  replicas: 3
  template:
    metadata:
      labels:
        name: bash-loop-rc
    spec:
      containers:
        - name: bash-loop
          image: ubuntu
          command: ["/bin/bash", "-c", "while true; do sleep 10; done"]
```

2. Теперь создадим контроллер репликации:

```
> kubectl create -f bash-loop-rc.yaml
replicationcontroller "bash-loop-rc" created
```

3. Вот результат его работы:

```
> kubectl get rc
NAME           DESIRED   CURRENT   READY   AGE
bash-loop-rc   3         3         3       1m
```

4. Как видите, желательное и текущее количества реплик равны 3, то есть запущены три пода. Давайте в этом убедимся:

```
> kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
bash-loop-rc-8h59t  1/1     Running   0          50s
bash-loop-rc-lsvtd  1/1     Running   0          50s
bash-loop-rc-z7wt5  1/1     Running   0          50s
```

5. Теперь можно создать контроллер автомасштабирования. Для того чтобы пример был более интересным, минимальное число реплик будет 4, а максимальное — 6:

```
> kubectl autoscale rc bash-loop-rc --min=4 --max=6 --cpu-percent=50
replicationcontroller "bash-loop-rc" autoscaled
```

6. Далее показан итоговый контроллер горизонтального автомасштабирования подов (можете использовать `hpa`). Здесь видна ссылка на контроллер репликации,

целевую и текущую загрузку процессора, а также минимальное и максимальное количества экземпляров подов. Имя совпадает с контроллером репликации, на который мы ссылаемся:

```
> kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
bash-loop-rc	bash-loop-rc	50 %	4	6	4	16m

7. Изначально контроллеру репликации были назначены три реплики, но в автомасштабировании минимальное число подов — 4. Как это повлияет на контроллер репликации? Правильно, желательное количество реплик выросло до четырех. Если средняя загруженность процессора превысит 50 %, этот показатель достигает 5 или даже 6:

```
> kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
bash-loop-rc	4	4	4	21m

8. Еще раз взглянем на поды и убедимся в том, что все работает. Обратите внимание на новый экземпляр пода (возрастом 17 минут), созданный в результате автомасштабирования:

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
bash-loop-rc-8h59t	1/1	Running	0	21m
bash-loop-rc-gjv4k	1/1	Running	0	17m
bash-loop-rc-lsvtd	1/1	Running	0	21m
bash-loop-rc-z7wt5	1/1	Running	0	21m

9. Если убрать горизонтальное автомасштабирование, контроллер репликации сохранит последнее желательное число реплик (в данном случае четыре). На этом этапе никто уже и не вспомнит, что он создавался с тремя репликами:

```
> kubectl delete hpa bash-loop-rc
```

```
horizontalpodautoscaler "bash-loop-rc" deleted
```

10. Как видите, контроллер репликации не был сброшен и все еще поддерживает работу четырех подов, даже без автомасштабирования:

```
> kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
bash-loop-rc	4	4	4	28m

Подойдем к вопросу с другой стороны. Что произойдет, если мы зададим горизонтальное автомасштабирование с диапазоном от 2 до 6 с той же целевой нагрузкой на процессор (50%)?

```
> kubectl autoscale rc bash-loop-rc --min=2 --max=6 --cpu-percent=50
replicationcontroller "bash-loop-rc" autoscaled
```

Контроллер репликации по-прежнему будет поддерживать четыре реплики, что соответствует заданному диапазону:

```
> kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
bash-loop-rc	4	4	4	29m

Однако в реальности нагрузка на процессор нулевая или близка к нулю. Количество реплик должно было уменьшиться до двух, но, поскольку контроллер горизонтального автомасштабирования не получил сведений о загрузенности процессора из Heapster, он не знает, что ему нужно уменьшить количество подов в контроллере репликации.

Плавающие обновления с автомасштабированием

Плавающие обновления — это краеугольный камень в обслуживании крупных кластеров. В Kubernetes они поддерживаются на уровне контроллера репликации и используют механизм развертывания. Однако такой подход несовместим с горизонтальным автомасштабированием. Дело в том, что во время плавающего развертывания создается новый контроллер репликации, а автомасштабирование привязано к старому. К сожалению, интуитивно понятная команда `kubectl rolling-update` инициирует обновление контроллера репликации.

Поскольку плавающие обновления так важны, я рекомендую всегда привязывать контроллеры автомасштабирования к объектам развертывания, а не к контроллерам репликации или наборам реплик. Благодаря этому контроллер автомасштабирования может указать количество реплик в спецификации развертывания и делегировать ему управление плавающими обновлениями и репликацией.

Далее показан конфигурационный файл, который мы использовали для развертывания сервиса `hue-reminders`:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
  template:
    metadata:
      name: hue-reminders
      labels:
        app: hue-reminders
    spec:
      containers:
        - name: hue-reminders
          image: g1g1/hue-reminders:v2.2
          ports:
            - containerPort: 80
```

Для обеспечения автомасштабирования и поддержания количества запущенных экземпляров в диапазоне от 10 до 15 можно создать конфигурационный файл `autoscaler`:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hue-reminders
```

```

namespace: default
spec:
  maxReplicas: 15
  minReplicas: 10
  targetCPUUtilizationPercentage: 90
  scaleTargetRef:
    apiVersion: v1
    kind: Deployment
    name: hue-reminders

```

Поле `kind` в разделе `scaleTargetRef` теперь имеет значение `Deployment` вместо `ReplicationController`. Это важно, так как у нас может быть контроллер репликации с таким же именем. Чтобы исключить совпадения и убедиться в том, что горизонтальное автомасштабирование привязано к нужному объекту, следует сделать поля `kind` и `name` согласованными.

Как вариант, воспользуйтесь командой `kubectl autoscale`:

```

> kubectl autoscale deployment hue-reminders --min=10--max=15
--cpu-percent=90

```

Ограничение ресурсов с помощью ЛИМИТОВ И КВОТ

Горизонтальное автомасштабирование создает поды на лету, поэтому нам нужно позаботиться об управлении ресурсами. Планировщик может легко выйти из-под контроля, реальной проблемой также является неэффективное использование ресурсов. Есть несколько факторов, которые могут влиять друг на друга неочевидным образом:

- ☐ общая мощность кластера;
- ☐ распределение ресурсов между узлами;
- ☐ распределение нагрузки между пространствами имен;
- ☐ контроллеры `DaemonSet`;
- ☐ контроллеры `StatefulSet`;
- ☐ принадлежность, непринадлежность, ограничения и доступ.

Прежде всего нужно понять ключевую проблему. При развертывании подов планировщик Kubernetes должен учитывать все эти факторы. В случае конфликтов или пересекающихся требований Kubernetes может оказаться сложно найти ресурсы для запуска новых подов. Возьмем, к примеру, экстремальный, но довольно простой случай, когда `DaemonSet` выполняет на каждом узле под, требующий 50 % доступной памяти. В результате Kubernetes не сможет развернуть ни один экземпляр пода, которому нужно больше половины памяти, поскольку у `DaemonSet` повышенный приоритет. Даже если выделить новые узлы, `DaemonSet` немедленно займет половину памяти.

Контроллеры `StatefulSet`, как и `DaemonSet`, требуют расширения количества узлов. Добавление новых участников в `StatefulSet` вызывается увеличением объема

данных, но в итоге дополнительные ресурсы берутся из общего пула Kubernetes. В мультиарендной конфигурации «недобросовестные соседи» способны вмешиваться в процесс создания узлов или выделения ресурсов. Можно тщательно расписать потребление в своем пространстве имен с учетом разных подов и их требований, однако на тех же физических узлах находятся соседние пространства имен, о содержимом которых вы, вероятно, даже не знаете.

Большинство этих проблем нивелируются за счет разумного использования квот на ресурсы и тщательного управления различными ресурсами кластера, такими как процессор, память и хранилище.

Включение квот на ресурсы

Большинство дистрибутивов Kubernetes поддерживают квоты на ресурсы «из коробки». API-сервер имеет флаг `--admission-control`, одним из аргументов которого должен быть `ResourceQuota`. Вам также нужно создать объект `ResourceQuota`. Kubernetes требует, чтобы в каждом пространстве имен было не больше одного такого объекта.

Типы квот

Существуют разные типы контролируемых квот. Они делятся на три категории: вычисление, хранение и объекты.

Квоты на вычислительные ресурсы

К вычислительным ресурсам относят процессор и память. Можно либо ограничивать их (`limit`), либо запрашивать какую-то их часть (`request`). Заметьте, что вместо `requests.cpu` и `requests.memory` можно указывать просто `cpu` и `memory`:

- ❑ `limits.cpu` — сумма лимитов на процессорное время во всех рабочих подах не превышает этого значения;
- ❑ `limits.memory` — сумма лимитов на память во всех рабочих подах не превышает этого значения;
- ❑ `requests.cpu` — общий объем процессорного времени, запрашиваемого рабочими подами, не превышает этого значения;
- ❑ `requests.memory` — общий объем памяти, запрашиваемой рабочими подами, не превышает этого значения.

Квоты на хранилище данных

Этот тип квот на ресурсы чуть более сложный. В каждом пространстве имен могут ограничиваться две вещи — объем хранилища и количество запросов на получение постоянного тома. Но помимо глобальных квот на доступное пространство и количества запрашиваемых постоянных томов, можно контролировать отдельные

классы хранилищ. Формат квот с указанием классов выглядит немного громоздким, но свою задачу выполняет:

- ❑ `requests.storage` — общий объем всех запрашиваемых постоянных томов не превышает этого значения;
- ❑ `persistentvolumeclaims` — максимальное количество запросов на выделение постоянных томов в заданном пространстве имен;
- ❑ `<storage-class>.storageclass.storage.k8s.io/requests.storage` — общий объем всех запрашиваемых постоянных томов класса `storage-class-name` не превышает этого значения;
- ❑ `<storage-class>.storageclass.storage.k8s.io/persistentvolumeclaims` — максимальное количество запросов на выделение постоянных томов класса `storage-class-name` в заданном пространстве имен.

Кроме того, в Kubernetes 1.8 появилась экспериментальная поддержка квот для временных хранилищ:

- ❑ `requests.ephemeral-storage` — общий объем запросов к временному локальному хранилищу не превышает этого значения;
- ❑ `limits.ephemeral-storage` — сумма лимитов на временное локальное хранилище не превышает этого значения.

Квота на количество объектов

Kubernetes поддерживает квоты еще на один вид ресурсов — API-объекты. Как мне кажется, это нужно для того, чтобы избавить API-сервер от необходимости управлять слишком большим количеством объектов. Не забывайте: платформа Kubernetes проделывает много внутренней работы, которой мы не видим. Ей часто приходится запрашивать сразу несколько объектов для аутентификации, авторизации и соблюдения множества различных политик. Простой пример — развертывание подов на основе контроллеров репликации. Представьте, что у вас миллиард объектов, управляющих репликацией, и всего три экземпляра пода, в результате большинство контроллеров не получают ни одной реплики. Тем не менее все ресурсы Kubernetes станут затрачиваться на проверку того, что ни один из миллиарда контроллеров не реплицировал свой шаблон пода и что им не нужно удалять никакие поды. Это довольно экстремальный пример, но суть должна быть ясна. Чем больше API-объектов, тем больше работы для Kubernetes.

Список избыточных объектов, от которых можно избавиться, немного неоднороден. Например, вы можете ограничить количество контроллеров репликации, но не наборов реплик, которые, по сути, представляют собой усовершенствованную версию тех же контроллеров и могут причинить идентичный ущерб, если их станет слишком много.

Но самым досадным упущением являются пространства имен. Все лимиты действуют в их рамках, но их количество нельзя ограничить. Таким образом, вы

легко перегрузите Kubernetes, создав слишком много пространств имен, даже если в каждом из них содержится небольшое количество API-объектов.

Далее перечислены все поддерживаемые объекты:

- ❑ **ConfigMaps** — максимальное количество карт конфигурации в одном пространстве имен;
- ❑ **PersistentVolumeClaims** — максимальное количество запросов на выделение постоянных томов в одном пространстве имен;
- ❑ **Pods** — максимальное количество рабочих подов в одном пространстве имен. Под считается рабочим, если его состояние `status.phase` равно `Failed` или `Succeeded`;
- ❑ **ReplicationControllers** — максимальное количество контроллеров репликации в одном пространстве имен;
- ❑ **ResourceQuotas** — максимальное количество квот на ресурсы в одном пространстве имен;
- ❑ **Services** — максимальное количество сервисов в одном пространстве имен;
- ❑ **Services.LoadBalancers** — максимальное количество сервисов балансировщика нагрузки в одном пространстве имен;
- ❑ **Services.NodePorts** — максимальное количество сервисов с привязкой к портам узла в одном пространстве имен;
- ❑ **Secrets** — максимальное количество объектов `secret` в одном пространстве имен.

Области действия квот

Некоторые ресурсы, например поды, могут находиться в разных состояниях, поэтому должна быть возможность учитывать это в квотах. Если у вас много подов, завершивших работу (что часто случается во время выкачивания обновлений), в создании новых экземпляров подов нет ничего плохого, даже если их общее количество превышает квоту. Чтобы этого добиться, квоту следует применять только к *рабочим* подам. Далее приводятся доступные области действия квот:

- ❑ **Terminating** — соответствует подам, у которых `spec.activeDeadlineSeconds >= 0`;
- ❑ **NotTerminating** — соответствует подам, у которых поле `spec.activeDeadlineSeconds` равно нулю;
- ❑ **BestEffort** — соответствует подам с негарантированным результатом работы;
- ❑ **NotBestEffort** — соответствует подам, которые всегда возвращают результат.

Область действия **BestEffort** относится только к подам, а области **Terminating**, **NotTerminating** и **NotBestEffort** могут применяться также к процессору и памяти. Это интересный момент, так как квота на ресурсы может предотвратить удаление пода. Далее перечислены поддерживаемые объекты:

- ❑ Cpu;
- ❑ limits.cpu;
- ❑ limits.memory;
- ❑ memory;
- ❑ pods;
- ❑ requests.cpu;
- ❑ requests.memory.

Запросы и ограничения

Запросы (request) и ограничения (limit) в контексте квот на ресурсы требуют явного задания целевого атрибута. Таким образом, платформа Kubernetes знает диапазон выделения ресурсов для каждого контейнера и может управлять общими квотами.

Работа с квотами

Сначала создадим пространство имен:

```
> kubectl create namespace ns
namespace "ns" created
```

Использование контекста конкретного пространства имен

Работая с любым пространством имен, кроме стандартного, я предпочитаю за-действовать *контексты*, чтобы не вводить `--namespace=ns` для каждой команды:

```
> kubectl config set-context ns --cluster=minikube --user=minikube --
namespace=ns
Context "ns" set.
> kubectl config use-context ns
Switched to context "ns".
```

Создание квот

1. Создадим объект `compute-quota`:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
spec:
  hard:
    pods: "2"
    requests.cpu: "1"
    requests.memory: 20Mi
    limits.cpu: "2"
```

```
limits.memory: 2Gi
> kubectl create -f compute-quota.yaml
resourcequota "compute-quota" created
```

2. Теперь добавим этот объект:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts-quota
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
> kubectl create -f object-count-quota.yaml
resourcequota "object-counts-quota" created
```

3. Мы можем просмотреть все квоты:

```
> kubectl get quota
NAME                                AGE
compute-resources                  17m
object-counts                      15m
```

4. Есть возможность также получить полную информацию с помощью команды describe:

```
> kubectl describe quota compute-quota
Name:                                compute-quota
Namespace:                           ns
Resource                               Used    Hard
-----
limits.cpu                            0        2
limits.memory                         0       2Gi
pods                                  0         2
requests.cpu                          0         1
requests.memory                       0       20Mi
> kubectl describe quota object-counts-quota
Name:                                object-counts-quota
Namespace:                           ns
Resource                               Used    Hard
-----
configmaps                           0        10
persistentvolumeclaims                0         4
replicationcontrollers                 0        20
secrets                               1        10
services                              0        10
services.loadbalancers                0         2
```

Этот вывод позволяет мгновенно сориентироваться в том, сколько важных глобальных ресурсов потребляется внутри кластера, не заостряя внимание на множестве отдельных объектов.

5. Добавим в пространство имен сервер NGINX:

```
> kubectl run nginx --image=nginx --replicas=1
deployment "nginx" created
> kubectl get pods
No resources found.
```

6. Как же так? Не найдено ни одного ресурса, хотя при создании объекта deployment не было никаких ошибок. Проверим ресурс развертывания:

```
> kubectl describe deployment nginx
Name:                nginx
Namespace:           ns
CreationTimestamp:    Sun, 11 Feb 2018 16:04:42 -0800
Labels:               run=nginx
Annotations:          deployment.kubernetes.io/revision=1
Selector:             run=nginx
Replicas:             1 desired | 0 updated | 0 total | 0 available | 1
unavailable
StrategyType:         RollingUpdate
MinReadySeconds:      0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Pod Template:
  Labels:  run=nginx
  Containers:
    nginx:
      Image:        nginx
      Port:         <none>
      Environment:  <none>
      Mounts:       <none>
      Volumes:      <none>
  Conditions:
    Type           Status    Reason
    ----           -
    Available       True      MinimumReplicasAvailable
    ReplicaFailure  True      FailedCreate
  OldReplicaSets:  <none>
  NewReplicaSet:   nginx-8586cf59 (0/1 replicas created)
  Events:
    Type      Reason      Age    From          Message
    ----      -
    Normal    ScalingReplicaSet  16m    deployment-controller    Scaled up replica set
nginx-8586cf59 to 1
```

Обратите внимание на раздел conditions. Состояние ReplicaFailure равно True, а в качестве причины указано FailedCreate. Как видите, при развертывании создан новый набор реплик под названием nginx-8586cf59, но сам под не создан, и мы все еще не знаем почему. Проверим набор реплик:

```
> kubectl describe replicaset nginx-8586cf59
Name:                nginx-8586cf59
Namespace:           ns
Selector:            pod-template-hash=41427915,run=nginx
Labels:              pod-template-hash=41427915
                    run=nginx
```

```

Annotations:    deployment.kubernetes.io/desired-replicas=1
                deployment.kubernetes.io/max-replicas=2
                deployment.kubernetes.io/revision=1
Controlled By:  Deployment/nginx
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Conditions:
  Type           Status      Reason
  ----           -
  ReplicaFailure  True       FailedCreate
Events:
  Type           Reason           Age             From              Message
  ----           -
  Warning        FailedCreate     17m (x8 over 22m) replicaset-controller
(combined from similar events): Error creating: pods "nginx-8586cf59-sdwxj"
is forbidden: failed quota: compute-quota: must specify limits.cpu,limits.
memory,requests.cpu,requests.memory

```

Вывод оказался очень длинным и занял несколько строк, но его содержание предельно ясно. В пространстве имен действует квота на вычислительные ресурсы, поэтому каждый контейнер должен ограничить процессорное время и память и вместе с тем запросить какое-то их количество. Чтобы обеспечить соблюдение квоты во всем пространстве имен, контроллер квот должен учитывать расход вычислительных ресурсов в каждом контейнере.

Итак, мы понимаем причину проблемы, но как ее решить? Можно создать отдельный объект `deployment` для каждого типа подов, которые будут использоваться, и тщательно продумать запросы и ограничения для процессора и памяти. Но что, если вы не совсем уверены? Что, если типов подов слишком много и вы не хотите следить за целой кучей конфигурационных файлов `deployment`?

В качестве альтернативы можно указать ограничение в командной строке во время выполнения развертывания:

```

> kubectl run nginx \
  --image=nginx \
  --replicas=1 \
  --requests=cpu=100m,memory=4Mi \
  --limits=cpu=200m,memory=8Mi \
  --namespace=ns

```

Это сработает, однако создание объектов `deployment` на лету со множеством аргументов — не самый надежный способ управления кластером:

```

> kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
nginx-2199160687-zkc2h             1/1     Running   0           2m

```

Использование диапазонов ограничений для вычислительных квот по умолчанию

1. Более предпочтительный вариант — задать вычислительные ограничения по умолчанию. Укажем диапазоны ограничений. Далее показан конфигурацион-

ный файл, который устанавливает некоторые стандартные значения для контейнеров:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
spec:
  limits:
  - default:
      cpu: 200m
      memory: 6Mi
    defaultRequest:
      cpu: 100m
      memory: 5Mi
    type: Container
> kubectl create -f limits.yaml
limitrange "limits" created
```

2. А здесь показаны текущие ограничения, действующие по умолчанию:

```
> kubectl describe limits limitsName: limits
Namespace: ns
Type Resource Min Max Default Request Default Limit Max
Limit/Request Ratio
-----
Container cpu - - 100m 200m -
Container memory - - 5Mi 6 Mi -
```

3. Теперь снова запустим NGINX, не указывая никаких запросов или ограничений для процессора и памяти. Но сначала удалим уже развернутый экземпляр NGINX:

```
> kubectl delete deployment nginx
deployment "nginx" deleted
> kubectl run nginx --image=nginx --replicas=1
deployment "nginx" created
```

4. Проверим, был ли создан под. Ответ положительный:

```
> kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-8586cf59-p4dp4 1/1 Running 0 16m
```

Выбор и регулирование мощности кластера

Такие возможности Kubernetes, как горизонтальное автомасштабирование, DaemonSet, StatefulSet и квоты, позволяют масштабировать и контролировать поды, хранилища и другие объекты. Однако в итоге мы все равно упираемся в физические (виртуальные) ресурсы, доступные кластеру. Если все узлы загружены на 100 %, нужно добавить узлов — тут уж ничего не поделаешь. В противном случае Kubernetes не сможет масштабироваться. В то же время, если

нагрузка постоянно колеблется, платформа Kubernetes способна уменьшить количество подов, но если вы при этом не уменьшите количество узлов, вам придется платить за излишнюю мощность. В облаке узлы можно останавливать и запускать заново.

Типы узлов

Самым простым решением будет выбрать единый тип узлов с заранее известными параметрами процессора, памяти и локального хранилища. Но этот вариант не самый оптимальный с точки зрения производительности и денежных затрат. Он упрощает планирование мощности, так как единственное, что вам остается определить, — общее количество узлов в кластере. С каждым новым узлом вы добавляете гарантированное количество процессорного времени и памяти, причем большинство кластеров Kubernetes и их компонентов имеют разные рабочие нагрузки. Это может быть система обработки потоков, в которой множество подов принимают какие-то данные и обрабатывают их в одном месте. Это дает большую нагрузку на процессор, но требования к памяти могут быть разными. Другие компоненты, такие как распределенный кэш, могут нуждаться в больших объемах памяти, не нагружая при этом процессор. В системах вроде Cassandra к каждому узлу необходимо подключать отдельный SSD-накопитель.

В любом из таких случаев узлы должны быть правильно помечены, чтобы платформа Kubernetes могла выбирать для их развертывания подходящие поды.

Выбор решений для хранения данных

Хранилище играет ключевую роль в масштабировании кластера. Существует три категории масштабируемых решений для хранения данных:

- ☐ собственное, выбранное вручную;
- ☐ облачное хранилище;
- ☐ решение, не входящее в состав кластера.

В первом случае вы устанавливаете в свой кластер некий вид хранилища. Это дает гибкость и полный контроль, однако управлением и масштабированием приходится заниматься самостоятельно.

Во втором варианте используется решение, предоставляемое облачной платформой. Вы получаете множество возможностей прямо «из коробки», но теряете контроль и, как правило, переплачиваете. К тому же в зависимости от сервиса можете оказаться жестко привязанными к конкретному провайдеру.

Третий вариант предполагает передачу больших объемов данных, что способно сказаться на производительности и денежных затратах. Обычно к этому решению прибегают в ситуациях, когда необходимо интегрироваться с уже существующей системой.

Конечно, большие кластеры способны иметь несколько хранилищ из разных категорий. Это одно из ключевых решений, которые предстоит принять. Помните, что ваши требования к хранилищу со временем могут измениться.

Компромисс между денежными затратами и временем отклика

Если для вас деньги не проблема, можете просто выделить для своего кластера избыточные ресурсы. Каждой узел получит лучшую из доступных аппаратных конфигураций, количество узлов будет больше, чем требуется для обработки существующей нагрузки, и в вашем распоряжении окажется огромное количество свободного дискового пространства. Загвоздка лишь в том, что деньги — это всегда проблема!

На избыточные ресурсы можно полагаться на начальных этапах, когда кластер принимает немного трафика. Вы можете запустить пять узлов, даже если большую часть времени достаточно лишь двух. Но умножьте это количество на 1000, и вас непременно спросят, зачем вам тысячи бездействующих компьютеров и петабайты пустого хранилища.

Ладно, допустим, после тщательных замеров и оптимизации все ваши ресурсы используются на 99,99999 %. Поздравляем, вы только что создали систему, неспособную выдержать малейшее повышение нагрузки или поломку одного узла без отклонения запросов или задержки ответов.

Нужно найти какой-то компромисс. Вы должны понять, как колеблется рабочая нагрузка, и подумать, какое соотношение затрат и выгоды по сравнению с сокращенным временем отклика или вычислительными возможностями дает избыточная мощность.

Если у вас есть жесткие требования к доступности и надежности, сделайте избыточность и выделение дополнительных ресурсов частью архитектуры системы. Представьте, что необходимо заменять компоненты на лету без простоя и незаметно для окружающих. Вероятно, потеря даже одной транзакции недопустима. В этом случае следует держать наготове резервные копии всех компонентов, а также дополнительные ресурсы, чтобы справиться с временными колебаниями нагрузки без специального вмешательства.

Эффективное использование узлов с разной конфигурацией

Для эффективного планирования мощности системы нужно понимать, как она себя ведет и какую нагрузку способен выдержать каждый компонент. Причем состоять она может из множества внутренних потоков данных. Имея четкое представление о предстоящих рабочих нагрузках, можно проанализировать рабочие процессы и попытаться понять, какая доля трафика приходится на ту или иную часть

системы. А затем вычислить количество подов и их требования к ресурсам. По своему опыту могу сказать, что нагрузки бывают фиксированными, динамическими, но предсказуемыми (например, в зависимости от рабочих часов) или совершенно хаотичными. Вы должны планировать с учетом всех типов нагрузки. Для этого поды можно распределять по узлам с разной конфигурацией в зависимости от конкретных требований.

Преимущества эластичных облачных ресурсов

Большинство облачных провайдеров поддерживают автоматическое масштабирование узлов, что идеально дополняет горизонтальное автомасштабирование подов в Kubernetes. Облачное хранилище тоже расширяется магическим образом, не требуя от вас никаких действий. Но и здесь есть свои нюансы, о которых следует знать.

Автомасштабирование узлов

Все крупные облачные провайдеры поддерживают автомасштабирование узлов. Есть некоторые различия, но увеличение и уменьшение масштаба в зависимости от загруженности процессора всегда доступно. Иногда предлагаются дополнительные критерии и балансировка нагрузки. Как видите, этим отчасти дублируются функции Kubernetes. Если облачный провайдер не поддерживает адекватного автомасштабирования с подходящим управлением, относительно легко реализовать его самостоятельно. Это позволит следить за потреблением ресурсов в кластере и обращаться к облачному API за добавлением или удалением узлов. Соответствующие критерии можно получить из Kubernetes.

Процесс добавления двух новых узлов на основе мониторинга загруженности процессора проиллюстрирован на рис. 9.2.

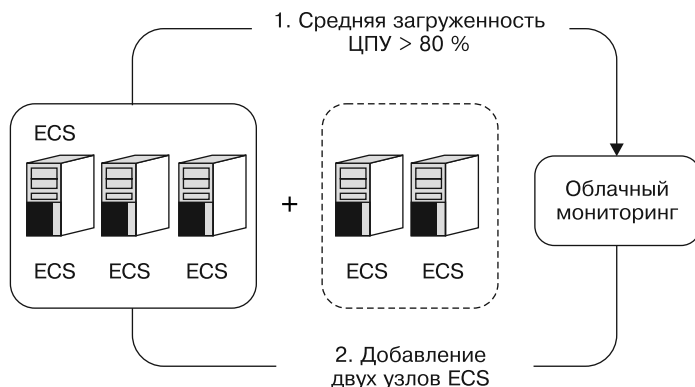


Рис. 9.2

Следите за своими облачными квотами

Во время работы с облачными провайдерами одним из самых раздражающих аспектов являются квоты. Я имел дело с четырьмя разными облачными платформами (AWS, GCP, Azure и Alibaba Cloud), и каждый раз квоты выходили мне боком. Квоты позволяют провайдерам планировать собственные мощности (а заодно защищают вас от случайного запуска миллиона узлов, которые вы никогда не сможете оплатить), но с точки зрения клиента это лишь еще один источник головной боли. Представьте, что вы разработали прекрасную систему автомасштабирования, которая работает самостоятельно, но при достижении отметки 100 узлов перестает повышать мощность. Вы быстро обнаруживаете, что доступны лишь 100 виртуальных машин, и обращаетесь в службу поддержки с просьбой увеличить квоту. Но пока запрос находится на рассмотрении, может пройти день или два. А в это время система будет неспособна выдерживать нагрузку.

Осторожно управляйте регионами

Облачные платформы разбиваются на регионы и зоны доступности. Некоторые услуги и узлы с определенной конфигурацией поддерживаются только в определенных регионах. Облачные квоты тоже учитываются на региональном уровне. Передавать данные внутри одного региона значительно быстрее и дешевле (а часто и вовсе бесплатно), чем между разными регионами. При проектировании кластера следует уделить особое внимание стратегии географического распределения. Если кластер должен охватывать разные регионы, вам, возможно, придется принимать непростые решения, касающиеся избыточности, доступности, производительности и финансовых затрат.

Hyper.sh и AWS Fargate в качестве альтернативы

Hyper.sh — это провайдер хостинга, в котором все запускается в виде контейнеров, а выделение аппаратных ресурсов происходит автоматически. Контейнеры стартуют за считанные секунды, и вам не нужно ждать, пока запустится новая виртуальная машина. *Hypernetes* — это разновидность Kubernetes с поддержкой *Hyper.sh*, она полностью устраняет проблемы с масштабированием узлов, так как концепции узла на уровне клиента просто не существует — есть только контейнеры (или поды).

В правой части схемы на рис. 9.3 показано, как Нурег-контейнеры выполняются непосредственно на «голом железе» в рамках *мультиарендной облачной платформы*.

Недавно компания Amazon представила систему AWS Fargate, которая похожим образом скрывает внутри себя работу с узлами, оставляя клиентам лишь возможность развертывать свои контейнеры в облаке. В сочетании с EKS она способна стать самой популярной платформой для развертывания Kubernetes.

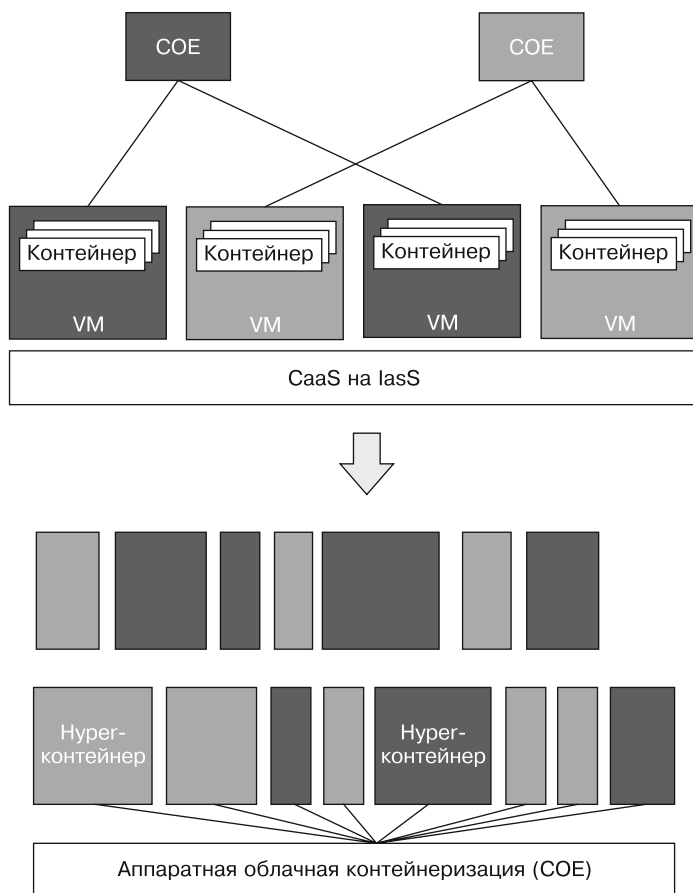


Рис. 9.3

Экстремальные нагрузки в Kubernetes

В этом разделе вы узнаете, как команда Kubernetes оптимизирует свое детище для экстремальных нагрузок. Цифры говорят сами за себя, но некоторые инструменты вроде Kubemark имеют интересные функции и даже могут пригодиться при тестировании кластера. В реальных условиях кластеры Kubernetes насчитывают до 3000 узлов. В CERN команда OpenStack достигла показателя 2 млн запросов в секунду: superuser.openstack.org/articles/scaling-magnum-and-kubernetes-2-million-requests-per-second.

В своей лаборатории компания Mirantis проверила производительность и масштабирование 5000 узлов Kubernetes (внутри виртуальных машин), развернутых на 500 физических серверах.



Больше информации о Mirantis вы найдете по адресу bit.ly/2oijqQY.

Компании OpenAI удалось развернуть свой кластер машинного обучения на основе Kubernetes на 2500 узлов. В результате было усвоено несколько полезных уроков: следует учитывать нагрузку, генерируемую агентами журналирования и их запросами, а события лучше хранить в отдельном кластере etcd (blog.openai.com/scaling-kubernetes-to-2500-nodes/).

В конце этого раздела вы по достоинству оцените, какие усилия и смекалка нужны для оптимизации Kubernetes для работы с большим количеством узлов. Вы узнаете, до каких пределов можно масштабировать один кластер Kubernetes и какой производительности при этом следует ожидать. Мы также рассмотрим некоторые инструменты и методики, которые помогут вам оценить скорость работы собственных кластеров.

Улучшение производительности и масштабируемости Kubernetes

В версии 1.6 команда Kubernetes уделила особое внимание производительности и масштабированию. Начиная с версии 1.2 в Kubernetes была заявлена поддержка кластеров размером до 1000 узлов, а в 1.6 этот показатель был доведен до невероятных 5000. Позже мы еще вернемся к цифрам, но сначала заглянем внутрь Kubernetes и попытаемся понять, за счет чего достигаются такие впечатляющие показатели.

Кэширование чтения в API-сервере

Kubernetes хранит состояние системы в etcd — это крайне надежное, но не очень скоростное решение (хотя в etcd3 внесены значительные улучшения, особенно в плане поддержки крупных кластеров). Различные компоненты Kubernetes работают со снимками данного состояния и не используют обновления в режиме реального времени, что позволяет улучшить пропускную способность за счет некоторого увеличения задержек. Раньше все снимки обновлялись watch-сервисами etcd. Теперь же для этого применяется кэширование чтения в памяти API-сервера. А уже этот кэш обновляется watch-сервисами. Такое нововведение значительно снизило нагрузку на etcd и повысило общую пропускную способность API-сервера.

Генератор событий жизненного цикла подов

Увеличение количества узлов в кластере — это ключ к горизонтальному масштабированию, важную роль играет также плотность размещения подов, то есть число

их экземпляров, которыми kubelet способен эффективно управлять на одном узле. Если плотность низкая, на одном узле нельзя разместить много подов. Это означает, что выбор узлов с более мощными процессорами может оказаться бессмысленным, так как kubelet все равно не справится с большим количеством экземпляров. Как вариант, разработчики могут пожертвовать стройностью своей архитектуры и сделать поды более крупными, способными выполнять больше работы. В идеале выбор масштаба задач для подов не должен зависеть от особенностей платформы. Команда Kubernetes очень хорошо это понимает и прилагает много усилий для увеличения плотности размещения.

В системе Kubernetes 1.1 официально рекомендовалось размещать на одном узле не больше 30 подов. Лично я выделял с данной версией 40 экземпляров на узел, но из-за этого сервис kubelet съедал лишнее процессорное время у рабочих заданий. В Kubernetes 1.2 этот показатель подскочил сразу до 100 экземпляров.

Раньше сервис kubelet постоянно опрашивал среды выполнения в контейнерах каждого пода. Это оказывало большое давление на контейнеры, и во время пиковых нагрузок возникали проблемы с надежностью (в частности, связанные с перерасходом процессорного времени). В качестве решения был предложен *генератор событий жизненного цикла подов* (Pod Lifecycle Event Generator, PLEG). Он получает состояние всех экземпляров подов и контейнеров и сравнивает его с предыдущим состоянием. Это делается лишь один раз. В результате сравнения PLEG знает, какие поды нуждаются в повторной синхронизации, и обращается только к ним. Благодаря такому изменению нагрузка на процессор со стороны kubelet и сред выполнения контейнеров упала в четыре раза. Также уменьшилось время опрашивания, что улучшило отзывчивость.

На схеме, представленной на рис. 9.4, сравнивается нагрузка на процессор для 120 подов в Kubernetes 1.1 и 1.2. Четырехкратное уменьшение сразу бросается в глаза.

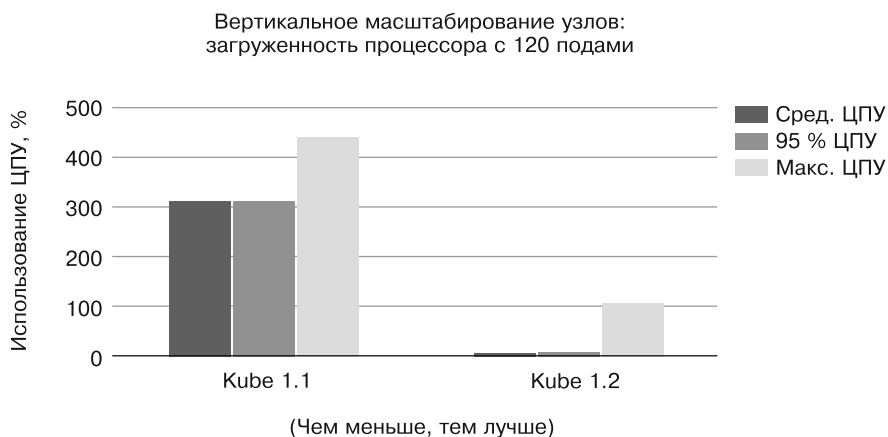


Рис. 9.4

Сериализация API-объектов с помощью протокольных буферов

API-сервер предоставляет REST API. В качестве формата сериализации в REST API обычно используется JSON, и API-сервер Kubernetes не исключение. Однако сериализация подразумевает преобразование между JSON и стандартными структурами данных. Это затратная операция. В крупномасштабных кластерах Kubernetes многим компонентам часто приходится получать или обновлять информацию в API-сервере. Расходы, необходимые для разбора и составления JSON, быстро накапливаются. В Kubernetes 1.3 разработчики добавили эффективный формат сериализации протокольных буферов. JSON никуда не делся, но все взаимодействие между внутренними компонентами Kubernetes происходит в новом формате.

etcd3

В версии 1.6 платформа Kubernetes перешла с etcd2 на etcd3. Это стало большим событием. Раньше из-за ограничений etcd2 (особенно из-за реализации watch-сервисов) масштабирование Kubernetes на 5000 узлов было невозможным. Данный факт послужил мотивацией для множества улучшений, появившихся в etcd3, так как команда CoreOS использовала Kubernetes для замеров производительности. Далее приведены несколько важнейших изменений.

- ❑ GRPC вместо REST. etcd2 работает с REST API, тогда как etcd3 поддерживает gRPC API (и REST API через шлюз gRPC). Протокол http/2, лежащий в основе gRPC, может применять одно TCP-соединение для множества потоков с запросами и ответами.
- ❑ Аренда вместо TTL. etcd2 использует *время жизни* (time to live, TTL) для истечения срока действия ключей. В etcd3 TTL применяется в сочетании с арендой, что позволяет назначать разным ключам один и тот же срок службы. Это значительно сокращает трафик, необходимый для поддержки постоянных соединений.
- ❑ Реализация watch-сервиса в etcd3 использует преимущества двунаправленных GRPC-потоков и поддерживает единое TCP-соединение для отправки разных событий. Это снижает расход памяти как минимум на порядок.
- ❑ Благодаря etcd3 платформа Kubernetes теперь хранит все свое состояние в формате protobuf. Это позволило избавиться от множества накладных расходов, связанных с сериализацией JSON.

Другие оптимизации

Разработчики Kubernetes выполнили много других оптимизаций.

- ❑ Выполнена оптимизация планировщика, что привело к увеличению производительности планирования в 5–10 раз.
- ❑ Все контроллеры переведены на новую рекомендуемую архитектуру с использованием общих информаторов, что снизило потребление ресурсов диспетчером

контроллеров (см. документ на <https://github.com/kubernetes/community/blob/master/contributors/devel/controllers.md>).

- ❑ Выполнена оптимизация отдельных операций в API-сервере (преобразования, глубокое копирование, заплатки).
- ❑ Уменьшено выделение памяти в API-сервере, что значительно сказалось на задержках API-вызовов.

Измерение производительности и масштабируемости Kubernetes

Чтобы улучшить производительность и масштабируемость, нужно иметь четкое представление о том, что именно нуждается в улучшении и каким образом будут оцениваться вносимые изменения. Вы также должны убедиться, что при этом в жертву не будут принесены основные свойства платформы и гарантии ее работоспособности. Улучшение производительности часто само собой приводит к улучшению масштабируемости, за что я его и люблю. Например, если вы снизите потребление процессорного времени в поде с 50 до 33 %, это будет означать, что на одном узле могут поместиться три пода вместо двух и общая масштабируемость внезапно выросла на 50 % (или же расходы сократились на 33 %).

SLO в Kubernetes

Kubernetes имеет *целевой уровень обслуживания* (Service Level Objectives, SLO). Это гарантии, которые следует соблюдать при попытке улучшить производительность и масштабируемость. В Kubernetes время отклика API-вызовов не должно превышать 1 с. Это 1000 мс. На самом деле в большинстве случаев оно оказывается на порядок меньше.

Измерение отзывчивости API

API имеют множество разных конечных точек, и их отзывчивость нельзя измерить каким-то одним простым числом. Каждый вызов следует оценивать отдельно. Кроме того, ввиду сложности и распределенности системы, не говоря уже о проблемах с сетью, результаты могут сильно варьироваться. Стандартная практика такова: произвести замеры для отдельных конечных точек, выполнить множество тестов и затем проанализировать относительные результаты.

Важно также предусмотреть достаточный объем аппаратных ресурсов для работы с большим количеством объектов. При тестировании ведущего узла команда Kubernetes использует 32-ядерную виртуальную машину со 120 Гбайт памяти.

На рис. 9.5 показаны 50, 90 и 99-й перцентили задержек разных важных API-вызовов в Kubernetes 1.3. Как видите, 90-й перцентиль очень маленький — меньше 20 мс. Даже 99-й перцентиль для операции удаления подов (DELETE) не превысил 125 мс, а средняя задержка для всех операций — около 100 мс.

Еще одной категорией API-вызовов являются операции LIST. Они более затратные поскольку для их выполнения необходимо собрать множество информации из

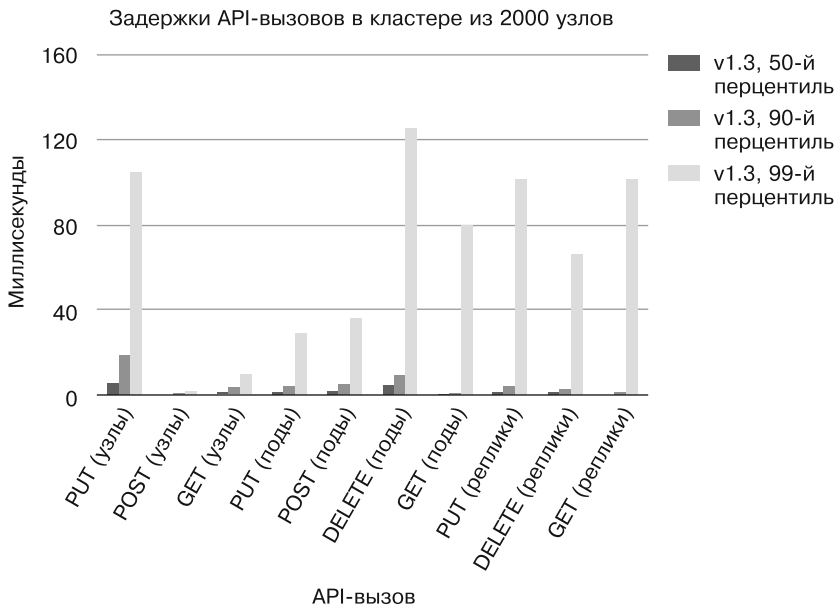


Рис. 9.5

всего кластера. Ответ, который в итоге формируется и отправляется, может быть довольно большим. Вот где по-настоящему проявляют себя такие способы оптимизации производительности, как кэширование чтения и сериализация протокольных буферов. Время отклика, естественно, больше, чем у одинарных API-вызовов, но требования SLO (1000 мс) удовлетворяются с запасом (рис. 9.6).

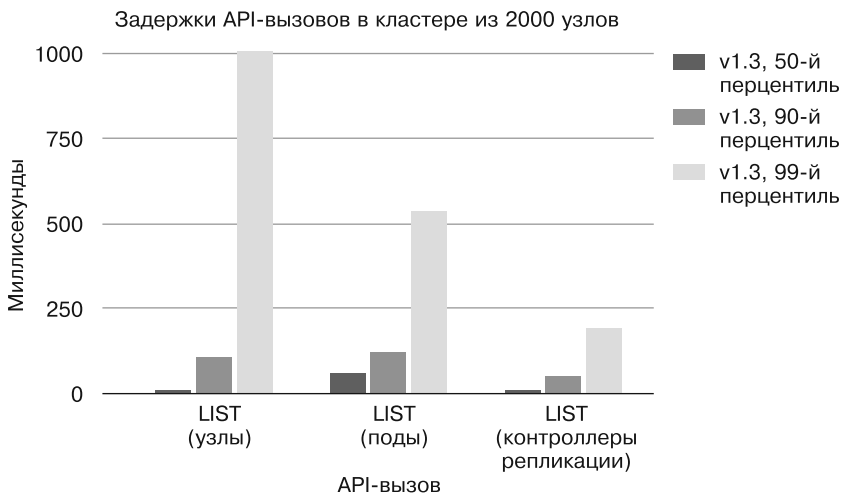


Рис. 9.6

Отличный результат. Но взгляните на задержку API-вызовов в кластере Kubernetes 1.6 с 5000 узлов (рис. 9.7).

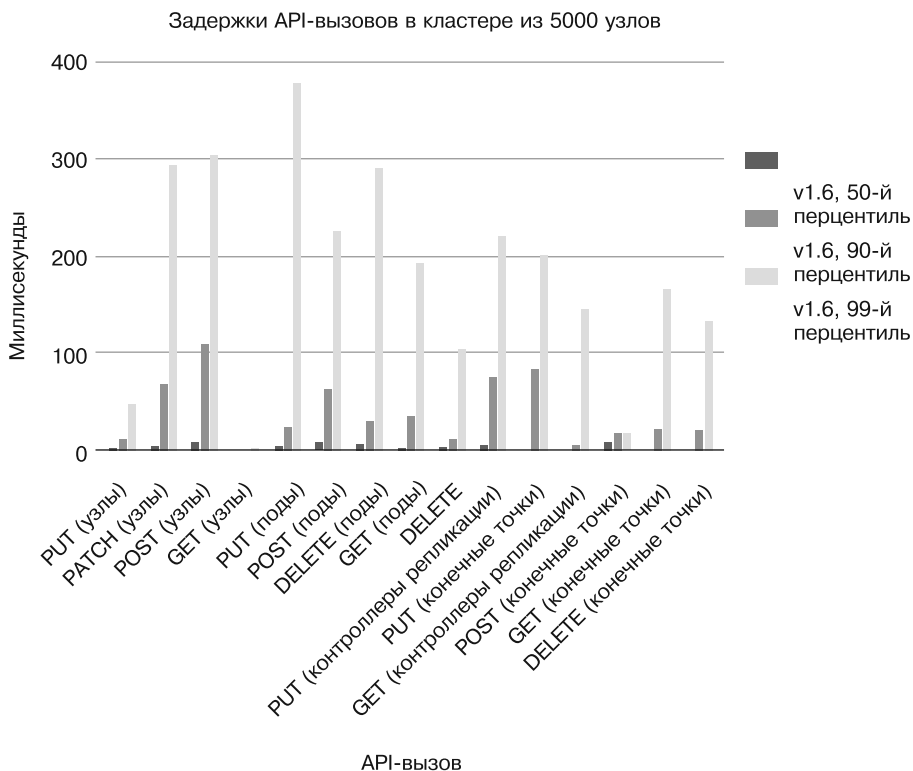


Рис. 9.7

Измерение полного времени запуска пода

Одна из самых важных характеристик производительности в крупном динамическом кластере — полное время запуска подов. В Kubernetes поды все время создаются, удаляются и перемещаются. Можно даже сказать, что планирование этих операций — основная функция Kubernetes.

На схеме (рис. 9.8) видно, что время запуска подов варьируется не так сильно, как отклик API-вызовов. Это логично, поскольку данная операция требует выполнения большого объема работы, которая не зависит от размера кластера, — например, запуска новой среды выполнения. В Kubernetes 1.2 с 1000 узлов 99-й перцентиль полного запуска пода был меньше 3 с. В Kubernetes 1.3 этот показатель снизился почти до 2,5 с. Поразительный факт: производительность

Kubernetes 1.3 в кластере с 2000 узлов даже немного лучше, чем в кластере с 1000 узлов.

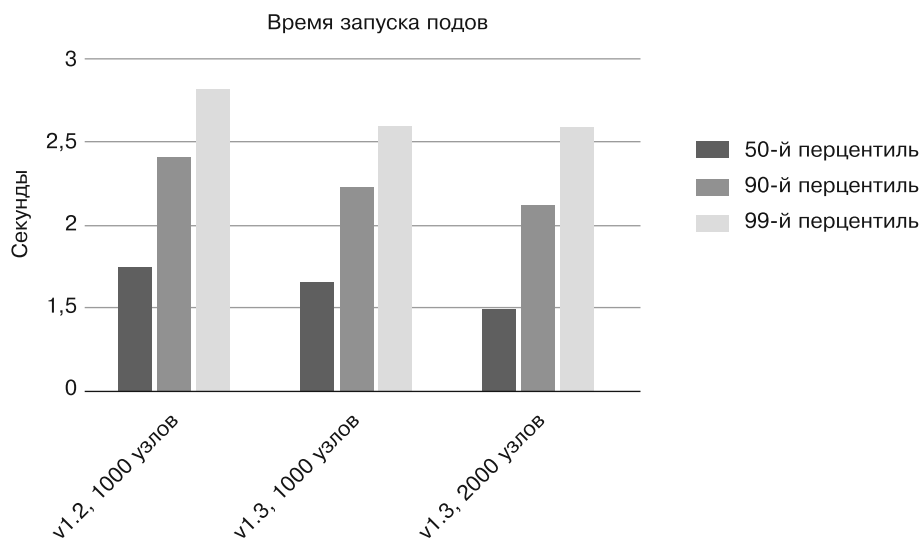


Рис. 9.8

Kubernetes 1.6 выводит производительность на новый уровень и еще лучше работает с большими кластерами (рис. 9.9).

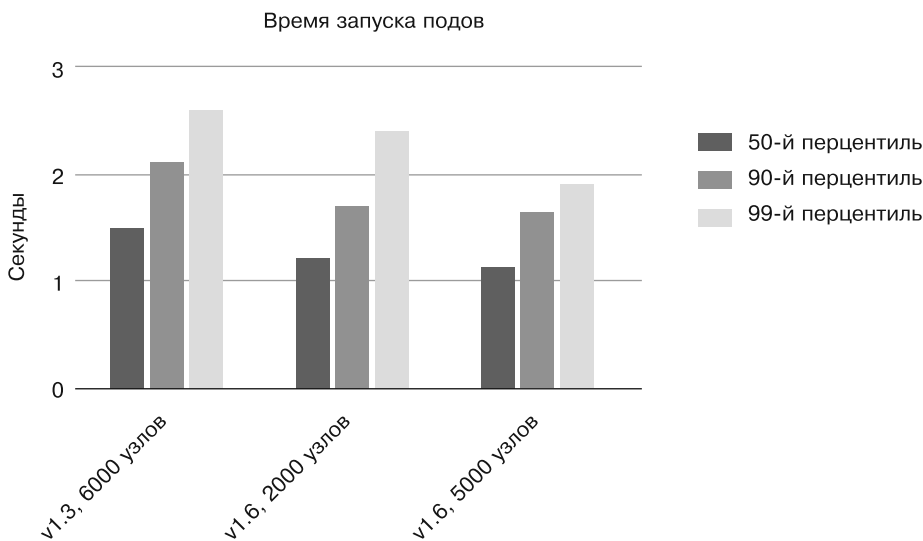


Рис. 9.9

Тестирование Kubernetes в крупномасштабных кластерах

Кластеры с тысячами узлов стоят дорого. Даже проекту Kubernetes, который получает помощь от Google и других гигантов индустрии, необходимо выработать разумный и экономный способ тестирования.

Команда разработчиков Kubernetes выполняет полноценное тестирование на настоящем кластере как минимум один раз для каждой новой версии. Это позволяет собрать реальные данные о производительности и масштабируемости. Но нужна также упрощенная и более дешевая методика, которая позволяла бы экспериментировать с потенциальными улучшениями и обнаруживать регрессии. Пришло время познакомиться с Kubemark.

Введение в Kubemark

Kubemark — это кластер Kubernetes с фиктивными (пустыми) узлами, предназначенный для упрощенной проверки производительности в крупномасштабных, но пустых кластерах. Некоторые компоненты Kubernetes, доступные в реальных узлах (например, kubelet), подменяются фиктивными аналогами. Фиктивная версия kubelet поддерживает множество функций оригинала, но не запускает никаких контейнеров и не подключает никаких томов. С точки зрения кластера Kubernetes все эти объекты существуют, и вы можете обращаться к ним через API-сервер. Чтобы сделать утилиту kubelet фиктивной, в нее внедряют клиент Docker, который не выполняет никакой работы.

Еще один важный пустой компонент — `hollow-proxy`, который подменяет собой Kuberпроху. В нем тоже задействуется оригинальный код, однако его реализация интерфейса прокси не делает ничего полезного и не трогает iptables.

Подготовка кластера Kubemark

Кластер Kubemark использует мощь Kubernetes. Чтобы подготовить его к работе, выполните следующие шаги.

1. Создайте обычный кластер Kubernetes, в котором можно запустить N узлов `hollow-node`.
2. Создайте отдельную ВМ для запуска всех ведущих компонентов Kubernetes.
3. Разверните поды на N узлов `hollow-node` в базовом кластере Kubernetes. Пустые узлы сконфигурированы для общения с API-сервером, запущенным на отдельной ВМ.
4. Создайте дополнительные поды, разверните их в базовом кластере и настройте для работы с API-сервером Kubemark.

Полное руководство для GCP находится по адресу bit.ly/2nPMkwc.

Сравнение Kubemark с настоящим кластером

По производительности кластеры Kubemark довольно похожи на настоящие. Разницей во времени запуска подов можно пренебречь. Различие в отзывчивости API более существенно, хотя обычно не больше чем в два раза. Но особенности поведения те же: улучшения/регрессии в настоящем кластере приводят к пропорциональным изменениям показателей Kubemark.

Резюме

В этой главе было рассмотрено много тем, связанных с масштабированием кластеров Kubernetes. Вы узнали, как горизонтальное автомасштабирование позволяет автоматически регулировать количество запущенных подов в зависимости от загрузки процессора и других показателей, как правильно и безопасно выкатывать плавающие обновления в условиях автомасштабирования и как ограничивать потребление ресурсов с помощью квот. Затем мы сосредоточились на планировании общей мощности кластера и управлении его физическими и виртуальными ресурсами. В конце был представлен реальный пример с масштабированием одного кластера Kubernetes на 5000 узлов.

Теперь у вас должно сложиться четкое понимание всех факторов, которые следует учитывать в условиях динамических и растущих нагрузок. В вашем распоряжении есть разные инструменты для планирования и реализации собственной стратегии масштабирования.

Следующая глава посвящена продвинутой организации сети. Сетевая модель Kubernetes основана на общем сетевом интерфейсе (Common Networking Interface, CNI), и ее поддерживают различные провайдеры.

10 Продвинутая организация сети в Kubernetes

В этой главе мы рассмотрим важный вопрос — организацию сети. Kubernetes как платформа оркестрации занимается управлением контейнерами и подами, запущенными на разных компьютерах (физических или виртуальных), и требует наличия отдельной сетевой модели. Вы познакомитесь со следующими темами.

- ❑ Сетевая модель Kubernetes.
- ❑ Стандартные интерфейсы, поддерживаемые в Kubernetes, — EXEC, Kubenet и CNI (с акцентом на последний).
- ❑ Различные сетевые решения, которые удовлетворяют требованиям Kubernetes.
- ❑ Сетевые политики и способы балансирования нагрузки.
- ❑ Написание собственного дополнения для CNI.

По прочтении этой главы вы будете понимать, как Kubernetes работает с сетью и какие решения доступны в сфере стандартных интерфейсов, реализации сети и балансирования нагрузки. При желании сможете даже написать собственное дополнение для CNI.

Сетевая модель Kubernetes

Сетевая модель Kubernetes основана на плоском адресном пространстве. Все поды в кластере способны видеть друг друга. У каждого экземпляра пода есть собственный IP-адрес. Нет нужды в настройке NAT. Кроме того, все контейнеры в одном поде используют его IP-адрес и могут взаимодействовать друг с другом локально. И хотя это довольно специфический подход, он значительно облегчает жизнь как разработчикам, так и администраторам. Особенно упрощает он процесс миграции традиционных сетевых приложений в Kubernetes. Под выступает в роли традиционного узла, а контейнер представляет собой классический процесс.

Взаимодействие между контейнерами внутри пода

Под всегда разворачивается на одном физическом или виртуальном узле. Это означает, что все его контейнеры выполняются на одной и той же системе и способны взаимодействовать между собой различными способами, например через файловую

систему, любой механизм ИРС или с применением локального сервера и заранее оговоренных портов. Опасности конфликта портов между разными подами не существует, так как каждый экземпляр имеет собственный адрес и использование контейнером локального сервера относится лишь к IP-адресу его пода. Таким образом, если контейнер 1 в поде 1 подключается к порту 1234, прослушиваемому контейнером 2 в том же поде, он не будет конфликтовать ни с одним контейнером в поде 2, который прослушивает порт 1234 и размещен на том же узле. Есть только один нюанс: при открытии портов на сервере следует с осторожностью относиться к тому, какие поды принадлежат конкретному узлу.

Взаимодействие между подами

В Kubernetes подам выделяются публичные IP-адреса, видимые другим участникам сети и не ограниченные лишь одним узлом. Поды могут общаться напрямую, минуя механизм сетевой адресации, туннели, прокси и прочие абстракции. Модель взаимодействия, основанная на известных портах, может обойтись без конфигурации. Внутренний IP-адрес пода совпадает с тем, который видят другие экземпляры подов (в рамках сети кластера, без доступа извне). Это означает, что стандартные механизмы именования и обнаружения, такие как DNS, не требуют дополнительной настройки.

Взаимодействие между подами и сервисами

Поды могут общаться между собой с помощью IP-адресов и заранее оговоренных портов, но для этого они должны знать адреса друг друга. В каждом кластере Kubernetes поды постоянно создаются и удаляются. Сервисы предоставляют очень полезный промежуточный уровень, он остается стабильным, в то время как экземпляры подов, которые на самом деле отвечают на запросы, непрерывно меняются. Кроме того, вы получаете автоматическую высокодоступную балансировку нагрузки, поскольку утилита kube-proxy на каждом узле отвечает за перенаправление трафика к подходящему поду (рис. 10.1).

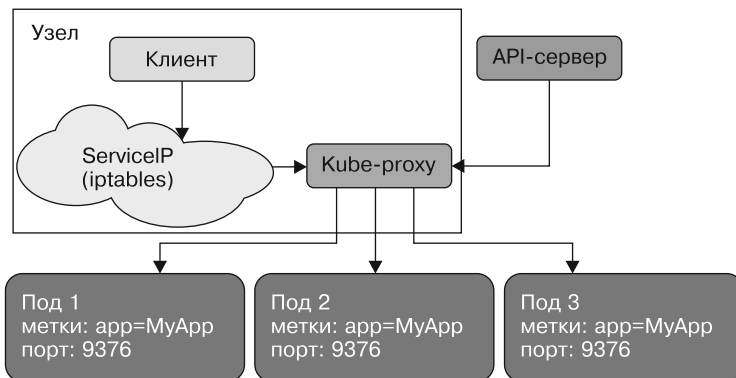


Рис. 10.1

Внешний доступ

Рано или поздно какой-нибудь контейнер потребует доступа извне. IP-адреса подов не видны за пределами сети кластера. Правильным выбором будет использовать сервисы, но при этом обычно приходится выполнять два перенаправления. Например, облачные балансировщики нагрузки знают о Kubernetes, поэтому не могут направить трафик непосредственно сервису на узле, который запускает под, способный обработать соответствующий запрос. Вместо этого трафик поступает в произвольный узел кластера, а kube-proxy перенаправляет его к подходящему экземпляру пода, если текущий узел не содержит нужного пода.

Как видно на рис. 10.2, внешний балансировщик нагрузки просто передает трафик всем узлам с утилитой kube-proxy, а та берет на себя дальнейшую маршрутизацию, если таковая потребуется.

Сетевые возможности Kubernetes и Docker

Docker использует другую модель работы с сетью, хотя со временем она стала тяготеть к той, которая применяется в Kubernetes. В Docker у каждого контейнера есть собственный частный IP-адрес в диапазоне 172.xxx.xxx.xxx, изолированный внутри родительского узла. Контейнеры, находящиеся на одном узле, могут общаться между собой с помощью адресов, относящихся к этому же диапазону. С точки зрения платформы Docker это вполне обоснованно, так как она не поддерживает концепцию подов с разными контейнерами внутри, которые могут взаимодействовать друг с другом. Таким образом, каждый контейнер представлен в виде облегченной ВМ с собственным сетевым идентификатором. В Kubernetes же контейнеры из разных подов не могут общаться локально, даже если находятся на одном узле (разве что через открытые порты, что нежелательно). Смысл заключается в том, что Kubernetes может удалять и создавать экземпляры подов где угодно, поэтому не стоит рассчитывать, что под будет доступен на каком-то конкретном узле. Заметным исключением являются наборы DaemonSet, но сетевая модель Kubernetes создана для работы в любых сценариях и не предусматривает специфические случаи, такие как прямое взаимодействие между разными подами на одном узле.

Как же Docker-контейнеры на разных узлах общаются между собой? Контейнер должен опубликовать порты своего сервера. Это требует координации, ведь если два контейнера опубликуют один и тот же порт, произойдет конфликт. Далее контейнеры (или другие процессы) подключаются к порту сервера, направленному к нужному контейнеру. Серьезный недостаток данного метода в том, что контейнеры не могут регистрироваться на внешних сервисах, потому что не знают IP-адрес своего узла. Такое ограничение можно обойти, передавая IP-адрес в виде переменной окружения при запуске контейнера, но это требует внешней координации и усложняет процесс.

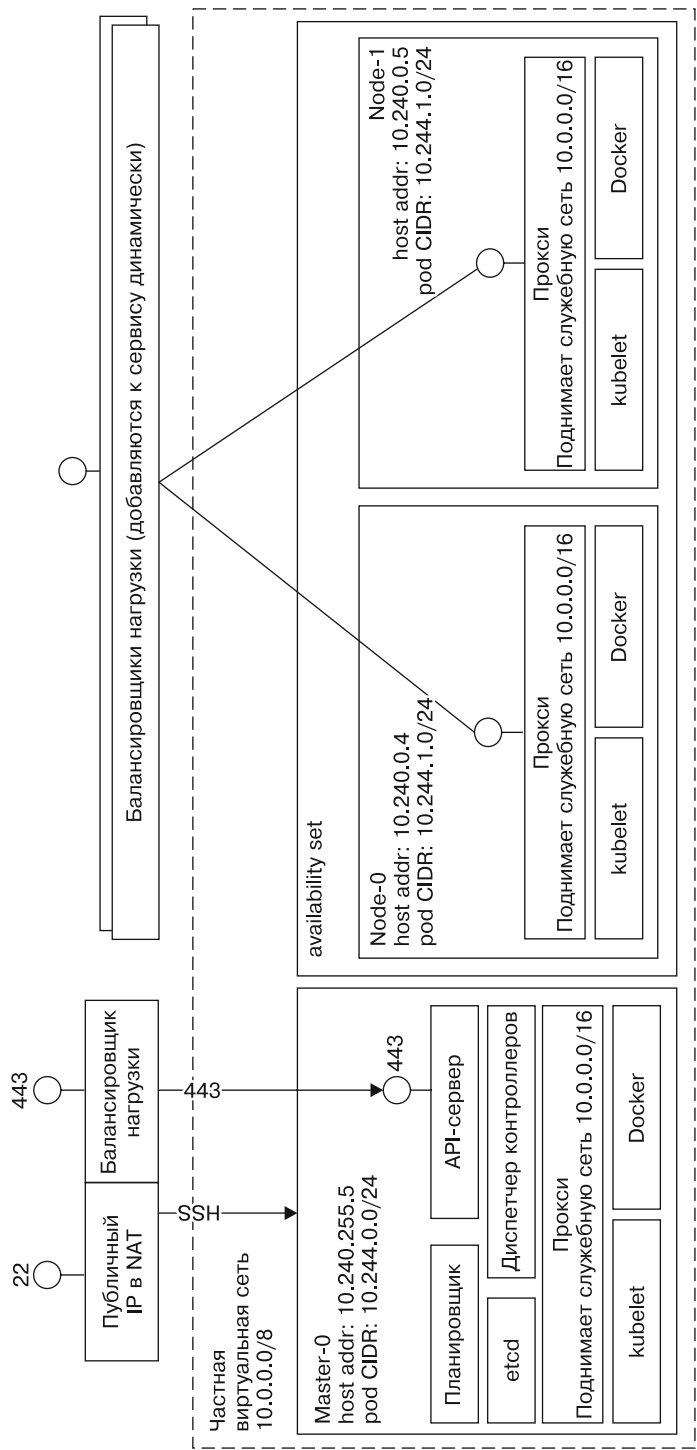


Рис. 10.2

На рис. 10.3 показана сетевая модель Docker. Каждый контейнер имеет IP-адрес, на каждом узле Docker создает мост `docker0`.

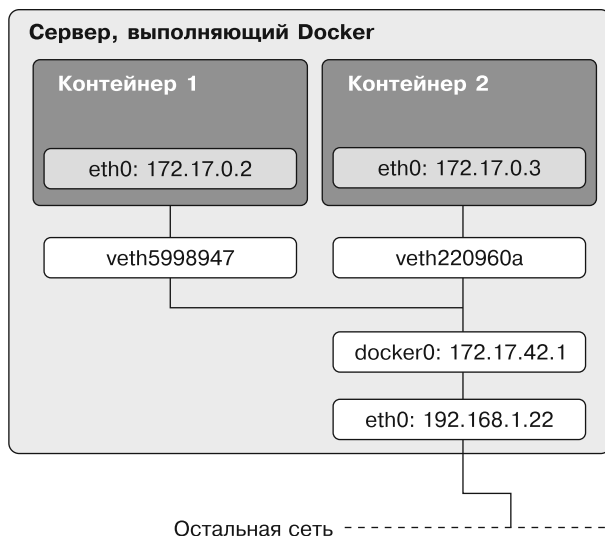


Рис. 10.3

Поиск и обнаружение

Чтобы поды и контейнеры могли взаимодействовать между собой, они сначала должны найти друг друга. Для определения чужого или публикации собственного местоположения существует несколько способов. Есть также шаблоны проектирования, позволяющие контейнерам общаться не напрямую. Каждый из этих вариантов имеет свои плюсы и минусы.

Самостоятельная регистрация

Этот способ уже упоминался несколько раз. Посмотрим, что он собой представляет. Когда контейнер запущен, ему известен IP-адрес его поды. Если он хочет быть доступным для других контейнеров в кластере, то может зарегистрировать свои IP-адрес и порт в некоем сервисе. Другие контейнеры будут обращаться к этому сервису за регистрационными данными для дальнейшего подключения. При плановом удалении контейнер аннулирует свою регистрацию. Если удаление происходит в нештатном режиме, нужен какой-то механизм, который будет отслеживать подобные ситуации. Например, сервис регистрации может периодически опрашивать все зарегистрированные контейнеры или же сами контейнеры могут время от времени отправлять сервису сообщения о том, что они все еще работают.

Одно из преимуществ самостоятельной регистрации состоит в том, что узко-специализированный регистрационный сервис берет на себя всю работу по отслеживанию контейнеров. Есть еще один плюс: контейнеры могут применять

сложные правила и временно отменять регистрацию, если из-за каких-то локальных условий они становятся недоступными, например, если контейнер занят и пока не хочет принимать новые запросы. Такой вид интеллектуального, децентрализованного, динамического балансирования нагрузки может оказаться очень сложным в реализации на глобальном уровне. Недостаток заключается в том, что придется добавлять еще один нестандартный компонент — сервис регистрации, о котором должны знать контейнеры.

Сервисы и конечные точки

Для регистрации можно задействовать и стандартные сервисы Kubernetes. Поды, принадлежащие сервису, регистрируются автоматически с помощью своих меток. Другие экземпляры подов могут искать их, обращаясь к конечным точкам или самому сервису (более распространенный подход). В последнем случае сообщения, переданные сервису, будут перенаправлены к одному из подходящих подов.

Слабая связанность с использованием очередей

Представьте, что контейнеры могут общаться между собой, не зная IP-адресов и портов друг друга или даже IP-адреса или сетевого имени сервиса. Что, если большинство коммуникаций сделать асинхронными и слабо связанными? Во многих случаях система может состоять из слабо связанных компонентов, которые зачастую не догадываются о существовании друг друга, не говоря уже об идентификаторах. Этого достигают за счет очередей. Компоненты (контейнеры) подписываются на сообщения из очереди, отвечают на них, приступают к работе и шлют уведомления о ходе ее выполнения, завершении и ошибках. Очереди имеют множество преимуществ.

- ❑ Повышение вычислительной мощности не требует координации, достаточно лишь добавить новые контейнеры, подписанные на очередь.
- ❑ Общую нагрузку можно легко оценить по глубине очереди.
- ❑ Благодаря наличию версий сообщений и каналов можно одновременно запускать разные версии одного и того же компонента.
- ❑ Благодаря наличию нескольких потребителей, обрабатывающих запросы в разных режимах, можно легко реализовать балансирование нагрузки и избыточность.

Очереди имеют следующие недостатки.

- ❑ Следует убедиться в том, что очередь имеет высокие устойчивость и доступность, иначе она может стать единой точкой отказа.
- ❑ Контейнеры должны работать с асинхронным API очереди (можно спрятать за абстракцией).
- ❑ Реализация запросов и ответов требует довольно громоздкого механизма подписки на очередь.

В целом, очереди — отличный выбор для крупномасштабных систем, они могут упростить координацию в сложных кластерах Kubernetes.

Слабая связанность с помощью хранилищ данных

Еще один метод слабого связывания основан на работе с базами данных наподобие Redis, которые хранят сообщения, доступные для других контейнеров. Такой подход вполне возможен, но базы данных в нем применяются не по прямому назначению, что часто приводит к реализации громоздких, нестабильных и не самых производительных решений. БД оптимизированы для хранения данных, но никак не для взаимодействия. Несмотря на это, их можно использовать в сочетании с очередями. Например, компонент может поместить какие-то данные в хранилище и сообщить об их готовности к обработке, отправив сообщение в очередь, другие компоненты получают это сообщение и начнут параллельную обработку данных.

Объекты Ingress в Kubernetes

Kubernetes предоставляет ресурс **Ingress** и контроллер, предназначенный для открытия доступа к внутренним сервисам извне. Все это можно реализовать самостоятельно, однако многие задачи, связанные с определением данного ресурса, свойственны большинству приложений с тем или иным типом доступа, таким как веб-сайты, CDN или системы защиты от DDoS-атак. Вы можете создавать собственные объекты **Ingress**.

Этот механизм часто используется для интеллектуального распределения нагрузки и принудительного разрыва TLS-соединений. Вместо подготовки и развертывания отдельного сервера NGINX можно воспользоваться встроенным ресурсом **Ingress**. Освежите свою память, вернувшись к главе 6, где обсуждались примеры применения этого ресурса.

Сетевые дополнения к Kubernetes

Работа с сетью может очень различаться и зависеть от предпочтений конкретных разработчиков. Поэтому Kubernetes имеет систему сетевых дополнений, гибкость которой позволяет учитывать любые сценарии. Мы уже подробно обсудили основное сетевое дополнение, CNI. Но Kubernetes предлагает более простую альтернативу под названием **Kubenet**. Прежде чем погружаться в детали, пройдемся по основным аспектам организации сети в Linux (это лишь вершина айсберга).

Основы работы с сетью в Linux

По умолчанию Linux имеет единое общее сетевое пространство, в котором доступны все аппаратные сетевые адаптеры. Но его можно разделить на несколько логических пространств, что крайне важно при работе с контейнерами.

IP-адреса и порты

Участников сети идентифицируют по IP-адресам. Серверы способны принимать входящие соединения на нескольких портах. Клиенты могут подключаться к серверам в своей сети (по TCP) или отправлять им сообщения (по UDP).

Сетевые пространства имен

Сетевые устройства, физически находящиеся в одной сети, можно разделять по разным пространствам имен. Таким образом, они смогут обращаться только к серверам, которые находятся в одном с ними пространстве. Соединение сетей или сетевых сегментов можно выполнить с помощью мостов, коммутаторов, шлюзов и маршрутизации.

Подсети, сетевые маски и бесклассовая адресация

Четкое разграничение сетевых сегментов очень помогает в проектировании и обслуживании сетей. Разделение сети на несколько мелких подсетей с общим префиксом — распространенная практика. Подсети можно определять в виде битовых масок, соответствующих их размеру (то есть тому, сколько сетевых узлов они способны вместить). Например, сетевая маска 255.255.255.0 означает, что первые три октета используются для маршрутизации, а количество отдельных узлов не может превышать 256 (в действительности 254). Для тех же целей часто применяют *бесклассовую адресацию* (Classless Inter-Domain Routing, CIDR), она более лаконична, кодирует больше информации и позволяет объединять сетевые узлы из нескольких устаревших классов (A, B, C, D, E). Например, 172.27.15.0/24 означает, что для маршрутизации отводятся первые 24 бита (три октета).

Виртуальные сетевые устройства

Виртуальные сетевые устройства (Virtual Ethernet, или veth) представляют физические сетевые адаптеры. Адаптер veth, а заодно и связанное с ним реальное устройство можно поместить в пространство имен, запретив тем самым прямой доступ к нему из других пространств, даже если другие устройства находятся в той же локальной сети.

Сетевые мосты

Сетевые мосты объединяют сетевые сегменты в агрегированную сеть, чтобы все узлы могли взаимодействовать друг с другом. Это делается на физическом (L1) и канальном (L2) уровнях сетевой модели OSI.

Маршрутизация

Объединяет отдельные сети. Обычно для этого используются таблицы маршрутизации, которые указывают сетевым устройствам, куда нужно направлять пакеты. Маршрутизация выполняется с помощью различных сетевых устройств, таких как маршрутизаторы, мосты, шлюзы, коммутаторы и брандмауэры, включая обычные серверы под управлением Linux.

MTU

MTU (maximum transmission unit) определяет максимальный размер пакетов. Например, в локальных сетях MTU равен 1500 байт. Чем больше MTU, тем лучше соотношение между полезным содержимым и заголовками (это хорошее свойство).

Недостаток — понижение минимальной задержки, поскольку вам приходится ждать передачи пакета целиком, более того, в случае сбоя весь пакет нужно будет передать заново.

Место пода в сети

На рис. 10.4 показаны отношения между подом, сервером и глобальной сетью с использованием интерфейса `veth0`.

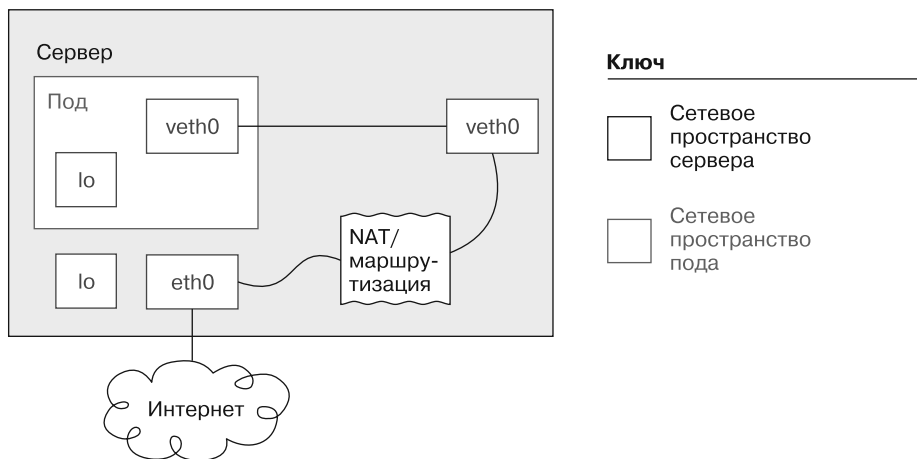


Рис. 10.4

Kubenet

Возвращаемся к Kubernetes. Kubenet — это крайне простое сетевое дополнение, которое создает для каждого пода стандартный для Linux мост `cbr0` и интерфейс `veth`. Облачные провайдеры обычно применяют его, чтобы сконфигурировать правила маршрутизации для взаимодействия между узлами (или для среды с одним узлом). Пара `veth` соединяет узел с каждым дочерним подом, используя IP-адреса из диапазона сервера.

Требования

Дополнение Kubenet предъявляет следующие требования.

- ☐ Узлу должна быть выделена подсеть, IP-адреса из которой будут назначаться подам.
- ☐ Начиная с версии 0.2.0, необходимо использовать стандартный мост CNI и дополнения `host-local`.
- ☐ Kubenet следует запускать с аргументом `network-plugin=kubenet`.
- ☐ Kubenet следует запускать с аргументом `--non-masquerade-cidr=<clusterCidr>`.

Задание MTU

MTU — это один из ключевых параметров производительности сети. Сетевые дополнения Kubernetes, такие как Kubelet, сами стараются вычислить оптимальный MTU, но иногда им требуется помощь. Если существующий сетевой интерфейс (например, мост `Docker docknet0`) установит небольшое значение MTU, Kubelet тоже будет его использовать. Еще один такой механизм — IPSEC: он требует понижения MTU из-за накладных расходов, вызванных инкапсуляцией, но Kubelet это не учитывает. Чтобы решить проблему, следует отказаться от автоматического вычисления MTU и вручную указать данный параметр. Для этого у каждого сетевого дополнения предусмотрен аргумент `--network-plugin-mtu`, хотя на сегодняшний день он учитывается только в Kubelet.

CNI

CNI (Container Networking Interface — сетевой интерфейс контейнеров) — это одновременно спецификация и набор библиотек для написания дополнений, которые конфигурируют сетевые интерфейсы в Linux-контейнерах (не только в Docker). На самом деле это производная чернового варианта спецификации gkt. CNI стремительно набирает популярность и находится в нескольких шагах от того, чтобы стать промышленным стандартом. Среди организаций, которые его используют, можно выделить следующие:

- ❑ Kubernetes;
- ❑ Kurma;
- ❑ Cloud foundry;
- ❑ Nuage;
- ❑ RedHat;
- ❑ Mesos.

Команда разработчиков CNI занимается поддержкой некоторых основных дополнений, но свой вклад в успех этого интерфейса вносят и многие сторонние проекты:

- ❑ **Project Calico** — виртуальная сеть третьего уровня;
- ❑ **Weave** — многоузловая сеть для Docker;
- ❑ **Contiv networking** — организация сети на основе политик;
- ❑ **Cilium** — BPF и XDP для контейнеров;
- ❑ **Multus** — поддержка нескольких сетевых интерфейсов в поде;
- ❑ **CNI-Genie** — общее сетевое дополнение для CNI;
- ❑ **Flannel** — реализация топологии «сетевая ткань» для контейнеров Kubernetes;
- ❑ **Infoblox** — управление IP-адресами контейнеров на промышленном уровне.

Среда выполнения контейнера

CNI описывает спецификацию дополнений для контейнеров с сетевыми возможностями, при этом дополнения должны подключаться к среде выполнения контейнера, которая предоставляет какие-то сервисы. С точки зрения CNI контейнер — это участник сети со своим собственным IP-адресом. Это справедливо для Docker, где IP-адрес назначается каждому контейнеру, однако в Kubernetes адресация происходит на уровне подов, а не контейнеров внутри них.

Аналогом подов являются rkt-контейнеры, которые могут содержать несколько Linux-контейнеров. Если вам это кажется запутанным, просто помните: CNI-контейнеры должны иметь собственные IP-адреса. Среда выполнения должна сконфигурировать сеть, запустить одно или несколько дополнений для CNI и передать им конфигурацию в формате JSON.

На рис. 10.5 показана среда выполнения контейнеров с несколькими дополнениями, которые взаимодействуют через интерфейс CNI.

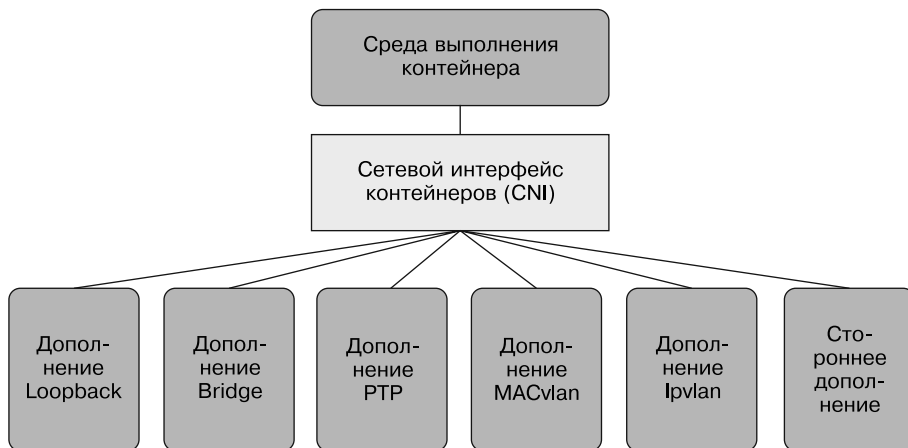


Рис. 10.5

Дополнение CNI

Задача дополнения CNI состоит в добавлении сетевого интерфейса в сетевое пространство контейнера и создании моста между ним и узлом поверх пары `veth`. Затем оно должно назначить IP-адрес через дополнение IPAM (диспетчер IP-адресов) и настроить маршруты.

Среда выполнения контейнеров (Docker, rkt или любая другая, удовлетворяющая CRI) запускает дополнение CNI в качестве исполняемого файла. Дополнение должно поддерживать следующие операции:

- ☐ добавление контейнера в сеть;
- ☐ удаление контейнера из сети;
- ☐ вывод версии.

Дополнение использует простой интерфейс командной строки, стандартные ввод/вывод и переменные окружения. Сетевая конфигурация в формате JSON передается дополнению через стандартный ввод. Остальные аргументы определяются в виде переменных окружения:

- ❑ CNI_COMMAND — обозначает нужную операцию, ADD, DEL или VERSION;
- ❑ CNI_CONTAINERID — идентификатор контейнера;
- ❑ CNI_NETNS — путь к файлу с описанием сетевого пространства;
- ❑ *CNI_IFNAME — название интерфейса, который нужно подготовить. Дополнение должно принять это название или вернуть ошибку (error);
- ❑ *CNI_ARGS — дополнительные аргументы, передаваемые пользователем во время запуска. Пары «ключ — значение» должны состоять из букв и цифр и разделяться точкой с запятой, например: FOO=BAR;ABC=123;
- ❑ CNI_PATH — список путей, по которым следует искать исполняемые файлы CNI-дополнений. Разделитель между путями зависит от ОС: в Linux это :, а в Windows — ;.

Если команда выполнена успешно, дополнение вернет нулевой код завершения, а сгенерированные интерфейсы (если это команда ADD) будут направлены в стандартный вывод в виде JSON. Этот незамысловатый интерфейс не привязан ни к одному конкретному языку программирования, компонентной архитектуре или двоичному формату. Авторы CNI-дополнений могут использовать любой язык на свой выбор.

Результатом запуска CNI-дополнения с командой ADD будет следующий код:

```
{
  "cniVersion": "0.3.0",
  "interfaces": [ (дополнения IPAM опускают этот ключ)
    {
      "name": "<имя>",
      "mac": "<MAC-адрес>", (требуется, если L2-адреса имеют значение)
      "sandbox": "<путь netns или идентификатор гипервизора>"
      (требуется для интерфейсов контейнера/гипервизора; в интерфейсах
      узлов опускается или остается пустым)
    }
  ],
  "ip": [
    {
      "version": "<4-или-6>",
      "address": "<ip-и-префикс-в-CIDR>",
      "gateway": "<ip-адрес-шлюза>", (не обязательно)
      "interface": <цифровой индекс в списке 'interfaces'>
    },
    ...
  ],
  "routes": [ (не обязательно)
    {
```

```

        "dst": "<ip-и-префикс-в-CIDR>",
        "gw": "<ip-следующего-перехода>" (не обязательно)
    },
    ...
]
"dns": {
    "nameservers": <список-dns> (не обязательно)
    "domain": <имя-локального-домена> (не обязательно)
    "search": <список-дополнительных-поисковых-доменов> (не обязательно)
    "options": <список-параметров> (не обязательно)
}
}

```

Сетевая конфигурация, подаваемая на ввод, содержит много информации: `cniVersion`, имя, тип, `args` (опционально), `ipMasq` (опционально), `ipam` и `dns`. Последние два параметра являются ассоциативными массивами с собственными ключами. Пример сетевой конфигурации приведен далее:

```

{
    "cniVersion": "0.3.0",
    "name": "dbnet",
    "type": "bridge",
    // зависит от типа (дополнения)
    "bridge": "cni0",
    "ipam": {
        "type": "host-local",
        // только для ipam
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1"
    },
    "dns": {
        "nameservers": [ "10.1.0.1" ]
    }
}

```

Вы также можете добавлять другие элементы, предусмотренные для конкретных дополнений. В нашем случае элемент `bridge: cni0` поддерживается только дополнением `bridge`.

Спецификация CNI также поддерживает списки сетевых конфигураций с упорядоченным запуском CNI-дополнений. Позже я покажу полноценную реализацию дополнения для CNI.

Сетевые решения для Kubernetes

Организация сети — обширная тема. Существует множество способов настроить сеть с подключением устройств, подов и контейнеров. Kubernetes вас в этом не ограничивает. Все, что предписывает данная платформа, — это высокоуровневая сетевая модель с плоским адресным пространством для подов. В рамках этого пространства можно реализовать множество хороших решений с разными возможностями и для разного окружения. В этом разделе рассмотрим некоторые

из них и попытаемся понять, каким образом они укладываются в сетевую модель Kubernetes.

Создание мостов в аппаратных кластерах

Самым простым окружением является кластер на «голом железе», который представляет собой обычную физическую сеть уровня L2. Для подключения контейнеров к такой сети можно использовать стандартный для Linux мост. Это довольно кропотливая процедура, требующая опыта работы с низкоуровневыми сетевыми командами Linux, такими как `brctl`, `ip addr`, `ip route`, `ip link`, `nsenter` и т. д. Реализацию такого решения можно начать со знакомства со следующим руководством: blog.oddbit.com/2014/08/11/four-ways-to-connect-a-docker/ (ищите раздел *With Linux Bridge devices*).

Contiv

Contiv — это сетевое дополнение общего назначения. Оно предназначено для подключения контейнеров через CNI и может использоваться вместе с Docker (напрямую), Mesos, Docker Swarm и, естественно, Kubernetes. Contiv занимается сетевыми политиками и частично дублирует аналогичный объект в Kubernetes. Далее перечислены некоторые из возможностей этого сетевого дополнения:

- ❑ поддержка CNM в `libnetwork` и спецификации CNI;
- ❑ механизм политик с богатыми возможностями, обеспечивающий безопасность и предсказуемое развертывание приложений;
- ❑ лучшая в своем роде производительность контейнеров;
- ❑ мультиарендность, изоляция и пересекающиеся подсети;
- ❑ интеграция с IPAM и обнаружение сервисов;
- ❑ широкий выбор физических топологий:
 - протоколы уровня 2 (VLAN);
 - протоколы уровня 3 (BGP);
 - оверлейные сети;
 - Cisco SDN (ACI);
- ❑ поддержка IPv6;
- ❑ масштабируемая политика и распределение маршрутов;
- ❑ интеграция с шаблонами приложений, включая следующие:
 - Docker-compose;
 - диспетчер развертывания Kubernetes;
 - распределение нагрузки на сервисы, встроенное в балансировщик микросервисов типа «*восток — запад*» (east — west);
 - изоляция трафика во время хранения, контроля доступа (например, `etcd/consul`), передачи по сети и управления.

Contiv имеет множество возможностей. Этот инструмент реализует широкий спектр задач и поддерживает различные платформы, поэтому я не уверен, станет ли он лучшим выбором для Kubernetes.

Open vSwitch

Open vSwitch — это зрелое решение для создания виртуальных (программных) коммутаторов, поддерживаемое многими крупными игроками на рынке. Система *Open Virtualization Network (OVN)* позволяет выстраивать различные виртуальные сетевые топологии. У нее есть специальное дополнение для Kubernetes, но его очень непросто настроить (см. руководство <https://github.com/openvswitch/ovn-kubernetes>). У дополнения Linen CNI меньше возможностей, но его конфигурация проходит намного легче: <https://github.com/John-Lin/linen-cni>. Структура Linen CNI показана на рис. 10.6.

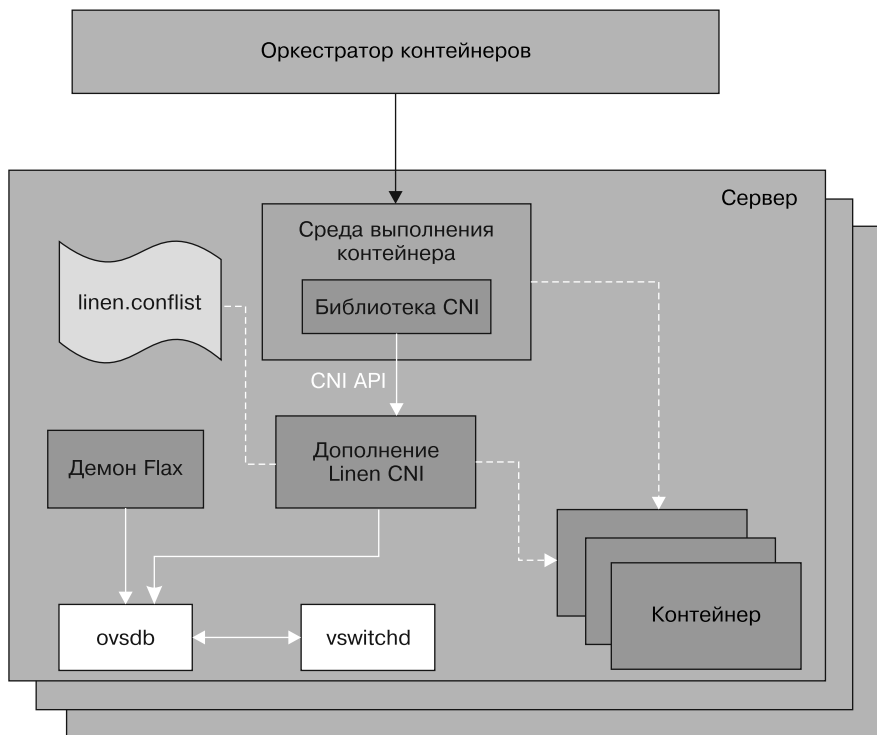


Рис. 10.6

Open vSwitch может объединять физические серверы, ВМ и поды/контейнеры в единую логическую сеть. Эта система поддерживает как оверлейный, так и физический режим.

Вот некоторые из ее ключевых возможностей:

- ❑ стандартная модель 802.1Q VLAN с магистральными и общедоступными портами;
- ❑ привязка по NIC с LACP или без него для коммутатора более высокого уровня;
- ❑ NetFlow, sFlow(R) и зеркалирование для обеспечения улучшенной видимости;
- ❑ конфигурация QoS (Quality of Service — качество обслуживания) плюс политики;
- ❑ туннелирование через Geneve, GRE, VXLAN, STT и LISP;
- ❑ контроль за разрывами соединения в 802.1ag;
- ❑ OpenFlow 1.0 плюс многочисленные дополнения;
- ❑ транзакционная база данных для хранения конфигурации с привязками для C и Python;
- ❑ высокопроизводительное перенаправление с помощью модулей ядра Linux.

Nuage Networks VCS

Virtualized Cloud Services (VCS) — это продукт от компании Nuage, который представляет собой хорошо масштабируемую, основанную на политиках платформу для построения *программно-определяемых сетей* (Software-Defined Networking, SDN). Это решение уровня предприятия, в основе которого лежат открытая система Open vSwitch (для перенаправления данных) и многофункциональный SDN-контроллер, построенный на открытых стандартах.

Платформа Nuage объединяет поды Kubernetes и сторонние окружения (виртуальные и аппаратные) в прозрачные оверлейные сети и позволяет описывать подробные политики для разных приложений. Ее механизм анализа в реальном времени дает возможность отслеживать видимость и безопасность приложений Kubernetes.

Кроме того, все компоненты VCS можно установить в виде контейнеров. Каких-либо специфических аппаратных требований нет.

Canal

Canal — это смесь двух проектов с открытым исходным кодом: Calico и Flannel. Отсюда и название. Проект Flannel, разрабатываемый командой CoreOS, занимается сетевыми возможностями контейнеров, а Calico отвечает за сетевые политики. Изначально они разрабатывались отдельно друг от друга, но пользователи хотели применять их совместно. Сейчас открытый проект Canal представляет собой шаблон развертывания для установки Calico и Flannel в виде отдельных CNI-дополнений. Компания *Tigera*, созданная основателями Calico, занимается поддержкой обоих проектов и даже планировала более тесную интеграцию, но с момента выпуска собственного решения для безопасной организации сети между

приложениями в Kubernetes приоритет сместился в сторону облегчения конфигурации и интеграции Flannel и Calico вместо разработки объединенного решения. На рис. 10.7 демонстрируется текущее состояние системы Canal и то, как она соотносится с платформами оркестрации, такими как Kubernetes и Mesos.

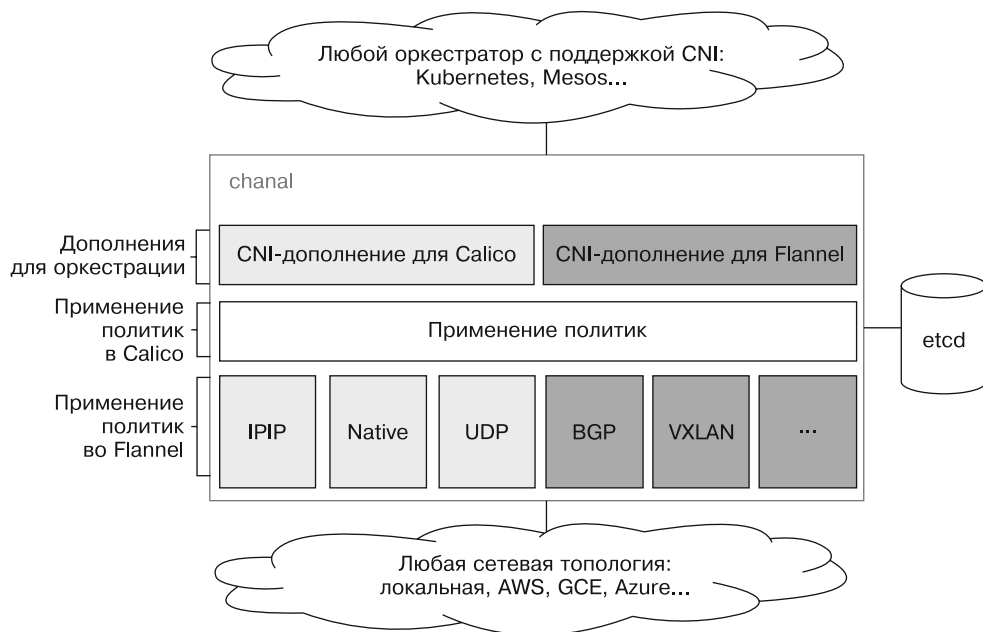


Рис. 10.7

Заметьте, что при интеграции с Kubernetes Canal обращается не напрямую к etcd, а к API-серверу Kubernetes.

Flannel

Flannel — это виртуальная сеть, которая выделяет каждому узлу по виртуальной сети для работы со средами выполнения контейнеров. На каждом узле запускается агент `flannel`, поднимающий подсеть на основе зарезервированного адресного пространства, хранящегося в кластере etcd. Обмен пакетами между контейнерами и, по большому счету, узлом производится одним из нескольких серверов. Чаще всего на сервере используется UDP поверх TUN-устройства, которое по умолчанию туннелирует трафик через порт 8285 (не забудьте открыть его в своем брандмауэре).

На рис. 10.8 подробно описываются различные компоненты сети Flannel, создаваемые ею виртуальные сетевые устройства и то, как они общаются с узлом и подом через мост `docker0`. Здесь также можно видеть процесс инкапсуляции UDP-пакетов и их перемещение между узлами.

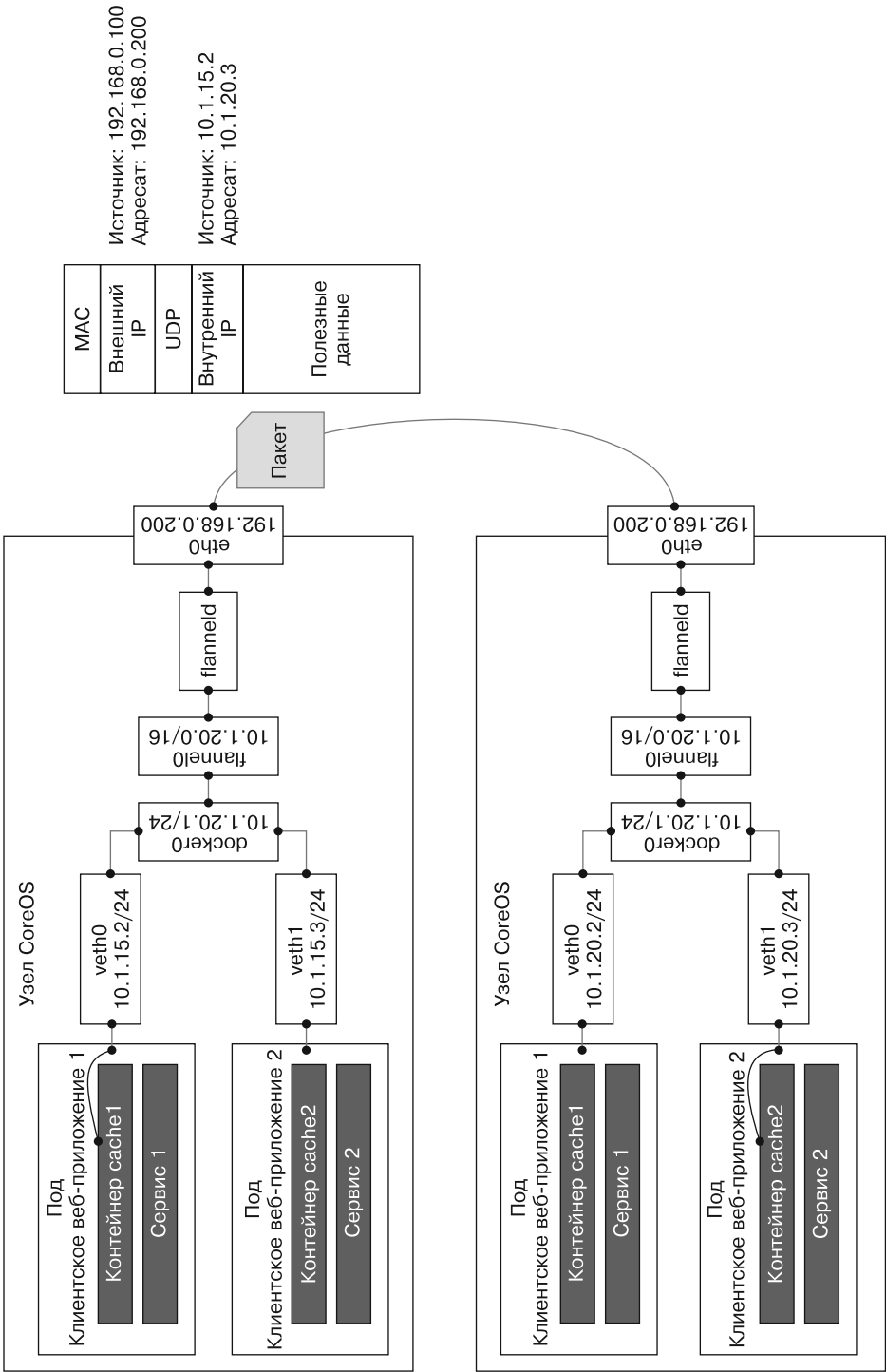


Рис. 10.8

Поддерживаются и другие сетевые технологии:

- ❑ `vxlan` — инкапсулирует пакеты с помощью VXLAN внутри ядра;
- ❑ `host-gw` — создает IP-маршруты к подсетям через IP-адреса удаленного сервера. Стоит отметить, что это требует прямого соединения на втором сетевом уровне между серверами, выполняющими Flannel;
- ❑ `aws-vpc` — создает IP-маршруты в таблице маршрутизации Amazon VPC;
- ❑ `gce` — создает IP-маршруты в сети Google Compute Engine;
- ❑ `alloc` — выполняет лишь выделение подсети, но не перенаправление пакетов;
- ❑ `ali-vpc` — создает IP-маршруты в таблице маршрутизации Alicloud VPC.

Проект Calico

Calico — это комплексное решение для организации сети между контейнерами и обеспечения сетевой безопасности. Его можно интегрировать со всеми основными платформами оркестрации и средами выполнения:

- ❑ Kubernetes (дополнение для CNI);
- ❑ Mesos (дополнение для CNI);
- ❑ Docker (дополнение для libnetwork);
- ❑ OpenStack (дополнение для Neutron).

Calico можно развертывать также локально или в публичном облаке с сохранением всех возможностей. Применение сетевых политик может зависеть от нагрузки, что обеспечивает четкий контроль трафика и гарантирует, что пакеты всегда будут доходить до нужных адресатов. Calico умеет автоматически импортировать сетевые политики с платформ оркестрации. На самом деле он отвечает за реализацию сетевых политик в Kubernetes.

Romana

Romana — это современное решение для организации сети между контейнерами. Оно изначально рассчитано на использование в облаке и работает на третьем сетевом уровне, опираясь на стандартные методы управления IP-адресами. Romana позволяет изолировать целые сети, создавая для них шлюзы и маршруты с помощью серверов на базе Linux. Работа на третьем сетевом уровне не требует инкапсуляции. Сетевая политика применяется ко всем конечным точкам и сервисам в виде распределенного брандмауэра. Romana облегчает локальное и гибридное развертывание между разными облачными платформами, так как больше не нужно настраивать виртуальные оверлейные сети.

Недавно появившиеся в Романа виртуальные IP-адреса позволяют локальным пользователям открывать доступ к своим сервисам в локальных сетях второго уровня, используя внешние адреса и спецификации сервисов.

Разработчики Romana утверждают, что их подход значительно улучшает производительность. На рис. 10.9 видно, как вместе с отказом от инкапсуляции VXLAN можно избавиться от множества накладных расходов.

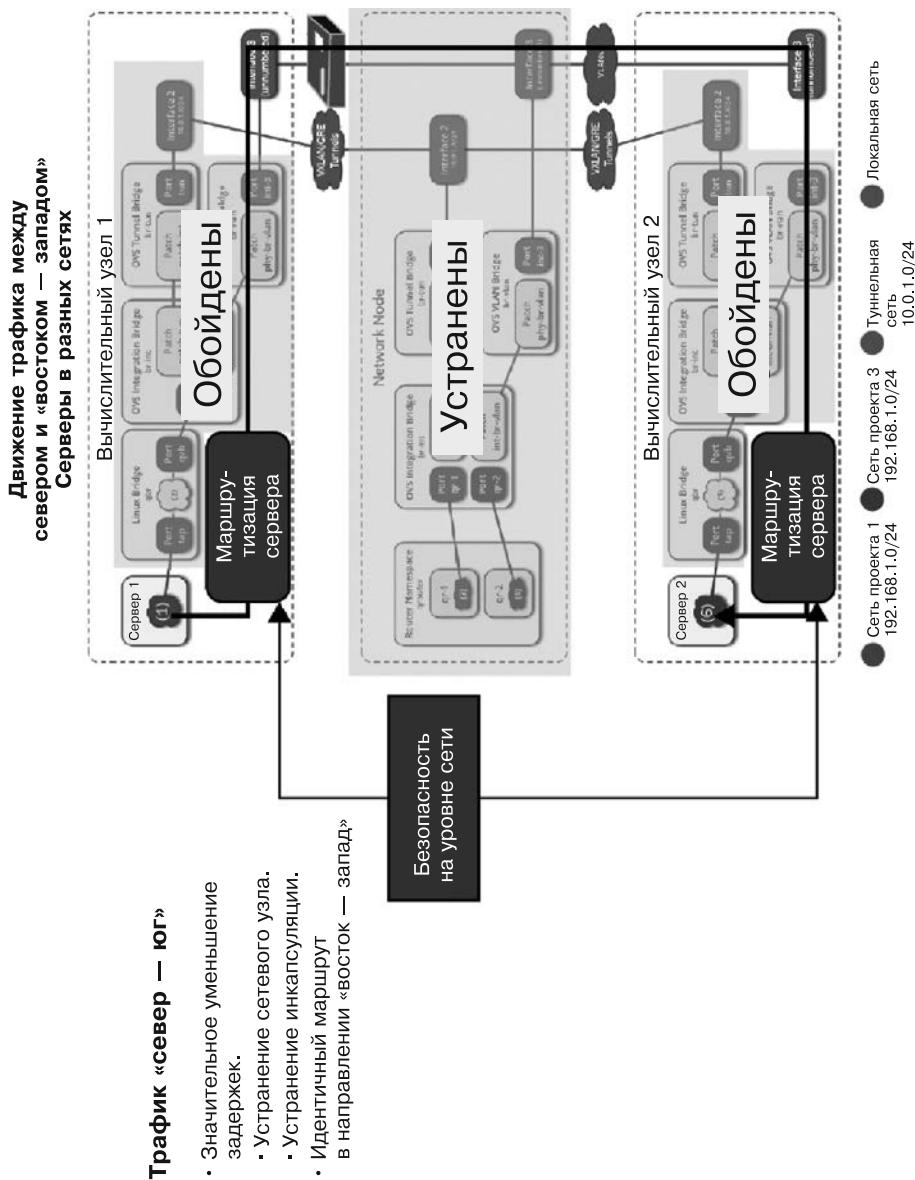


Рис. 10.9

Weave Net

Основными чертами проекта Weave Net являются простота в применении и отсутствие конфигурации. Он использует инкапсуляцию VXLAN и устанавливает микро-DNS на каждый узел. Как разработчик, вы будете иметь дело с высоким уровнем абстракции. После того как вы дадите имена своим контейнерам, Weave Net позволит вам подключаться к стандартным портам и задействовать соответствующие сервисы. Это помогает при миграции существующих приложений на платформы микросервисов и контейнеризации. В Weave Net предусмотрено CNI-дополнение для работы с Kubernetes и Mesos. Начиная с Kubernetes 1.4, интеграцию с Weave Net можно выполнить одной командой, которая разворачивает DaemonSet:

```
kubect1 apply -f https://git.io/weave-kube
```

Поды Weave Net, размещенные на каждом узле, отвечают за подключение любых других экземпляров подов к сети Weave. Weave Net поддерживает API с сетевыми политиками, предоставляя полноценное и простое в настройке решение.

Эффективное использование сетевых политик

Сетевая политика Kubernetes предназначена для управления трафиком, который направлен к определенным подам и пространствам имен. При управлении сотнями развернутых микросервисов (как часто бывает с Kubernetes) организация сетевых соединений между подами выходит на первый план. Важно понимать, что этот механизм лишь опосредованно относится к безопасности. Если злоумышленник способен проникнуть во внутреннюю сеть, он, скорее всего, сможет создать собственный экземпляр пода, который будет соответствовать сетевой политике и позволит свободно общаться с другими подами. В предыдущем разделе мы рассмотрели различные решения для организации сети в Kubernetes, сосредоточившись на сетевых интерфейсах. Здесь же основное внимание уделим сетевой политике, реализуемой поверх этих решений, хотя оба компонента тесно взаимосвязаны.

Архитектура сетевой политики в Kubernetes

Сетевая политика определяет, каким образом подмножества подов могут взаимодействовать друг с другом и с другими конечными точками сети. Ресурс `NetworkPolicy` использует метки для выбора подов и определяет список разрешительных правил, которые позволяют направлять трафик к выбранным экземплярам подов (в дополнение к тому, что уже позволено политикой изоляции в заданном пространстве имен).

Сетевые политики и CNI-дополнения

Сетевые политики и CNI-дополнения находятся в сложных отношениях. Одни дополнения реализуют как сетевые соединения, так и политику. Другие занимаются лишь одним из этих направлений. Они могут взаимодействовать с CNI-дополнением, которое реализует недостающий аспект (как, например, Calico и Flannel).

Конфигурация сетевых политик

Сетевые политики настраиваются с помощью ресурса `NetworkPolicy`. Вот как это выглядит:

```
apiVersion: networking.k8s.io/v1kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: awesome-project
      - podSelector:
          matchLabels:
            role: frontend
    ports:
      - protocol: tcp
        port: 6379
```

Реализация сетевых политик

Сам по себе API сетевых политик универсален и входит в состав Kubernetes API, однако его реализация тесно связана с сетевым решением. Это означает, что на каждом узле разворачивается специальный агент, или «страж», который:

- ❑ перехватывает весь трафик, поступающий в узел;
- ❑ проверяет, соответствует ли тот сетевой политике;
- ❑ перенаправляет или отклоняет каждый отдельный запрос.

Kubernetes предоставляет механизм для определения и хранения сетевых политик с помощью API. Применением сетевой политики занимается сетевое решение

или отдельный механизм, который тесно с ним интегрирован. Хорошим примером такого подхода являются Calico и Canal. Calico имеет собственные решения для организации сети и сетевых политик, которые работают вместе, однако использование последних можно реализовать поверх Flannel как часть Canal. В обоих случаях эти два аспекта тесно связаны между собой. На рис. 10.10 показаны контроллер для управления сетевыми политиками в Kubernetes и агенты, которые применяют их на отдельных узлах.

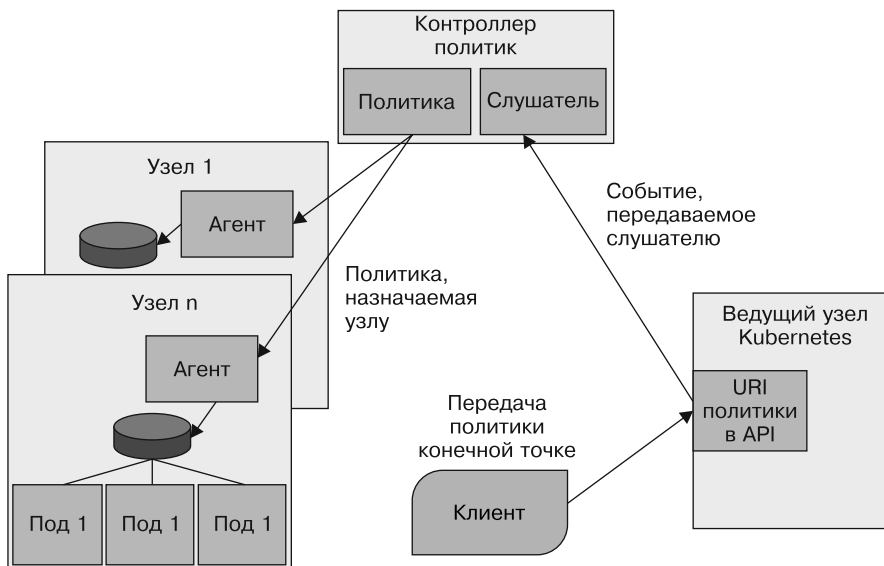


Рис. 10.10

Способы балансирования нагрузки

Балансирование нагрузки — это важнейший аспект динамических систем, к которым относится кластер Kubernetes. Узлы, виртуальные машины и поды приходят и уходят, но клиенты не могут следить за тем, кто именно обслуживает их запросы. Даже если бы это было возможно, вам пришлось бы заниматься такими непростыми задачами, как постоянное обновление динамической карты кластера и управление отключившимися, неотзывчивыми или просто медленными узлами. Балансирование нагрузки — проверенный временем и понятный всем механизм, который скрывает все эти внутренние перипетии от клиентов или внешних потребителей за счет добавления еще одного промежуточного слоя. Балансировщики нагрузки бывают внешними и внутренними. Их можно использовать совместно. Такой гибридный подход имеет свои плюсы и минусы, например понижение производительности в угоду гибкости.

Внешний балансировщик нагрузки

Внешний балансировщик нагрузки работает за пределами кластера Kubernetes. Он должен предоставляться сторонним провайдером и позволять настраивать проверку работоспособности, правила брандмауэра и получение его внешнего IP-адреса.

На рис. 10.11 показана связь между балансировщиком нагрузки (в облаке), API-сервером Kubernetes и узлами кластера. Внешний балансировщик всегда знает, на каких узлах запущены те или иные поды, и может распределять внешний служебный трафик между подходящими экземплярами подов.

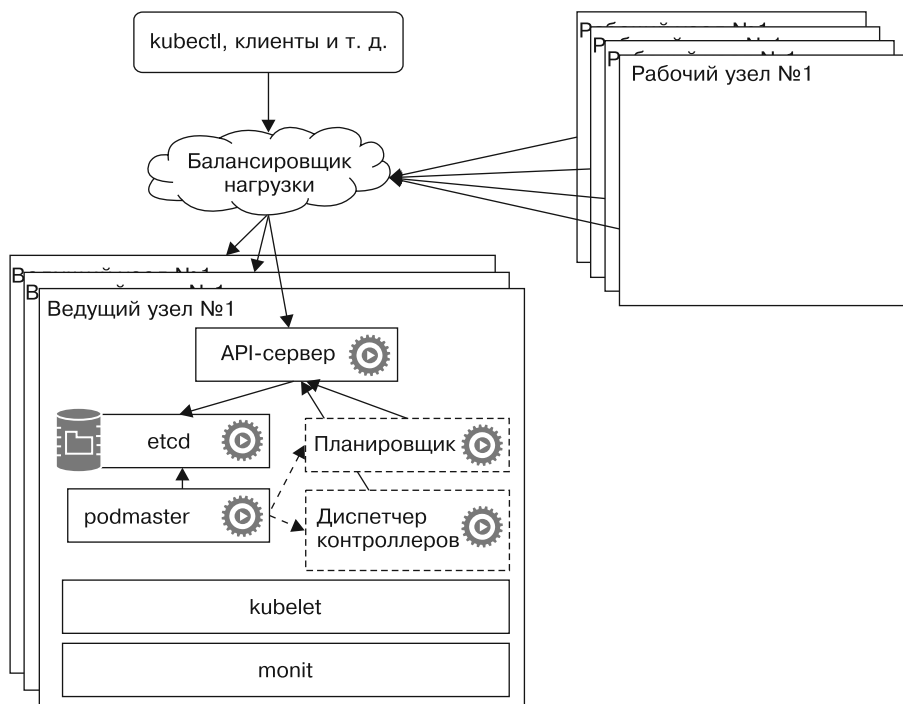


Рис. 10.11

Настройка внешнего балансировщика нагрузки

Внешний балансировщик нагрузки настраивается либо с помощью служебного конфигурационного файла, либо напрямую через `kubectl`. В качестве типа сервиса используется `LoadBalancer`, а не `ClusterIP`, который назначает балансировщиком нагрузки один из узлов Kubernetes. Для этого провайдер внешнего балансировщика должен быть правильно установлен и настроен в рамках кластера. Самый проверенный провайдер — это GKE от Google, однако другие облачные платформы тоже предоставляют интегрированные решения поверх своих облачных балансировщиков.

С помощью конфигурационного файла

Вот пример конфигурационного файла сервиса, который справляется с этой задачей:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "example-service"
  },
  "spec": {
    "ports": [{
      "port": 8765,
      "targetPort": 9376
    }],
    "selector": {
      "app": "example"
    },
    "type": "LoadBalancer"
  }
}
```

С помощью kubectl

Того же результата можно добиться напрямую с использованием команды `kubectl`:

```
> kubectl expose rc example --port=8765 --target-port=9376 \
--name=example-service --type=LoadBalancer
```

Выбор между `kubectl` и конфигурационным файлом `service` обычно зависит от того, как вы настраиваете остальную часть инфраструктуры и развертываете свою систему. Конфигурационные файлы более декларативны и, считается, лучше подходят для промышленных условий, в которых управление инфраструктурой должно быть воспроизводимым и проверяемым.

Поиск IP-адресов балансировщика нагрузки

Балансировщик нагрузки имеет два интересующих нас IP-адреса. Внутренний адрес можно использовать внутри кластера для доступа к сервису. Клиенты, не входящие в кластер, задействуют внешний адрес. Создание DNS-записи для внешнего IP-адреса — рекомендованный способ. Для получения обоих адресов применяется команда `kubectl describe`: внутренний находится в поле `IP`, а внешний обозначен как `LoadBalancer Ingress`.

```
> kubectl describe services example-service
Name: example-service
Selector: app=example
Type: LoadBalancer
IP: 10.67.252.103
LoadBalancer Ingress: 123.45.678.9
```

```

Port: <unnamed> 80/TCP
NodePort: <unnamed> 32445/TCP
Endpoints: 10.64.0.4:80,10.64.1.5:80,10.64.2.4:80
Session Affinity: None
No events.

```

Сохранение клиентских IP-адресов

Иногда сервис может быть заинтересован в исходных IP-адресах клиентов. В Kubernetes данная информация доступна только с версии 1.5. Пока что эта возможность находится в стадии бета-тестирования, она совместима только с GKE и основана на аннотациях. В Kubernetes 1.7 возможность сохранения исходного клиентского IP-адреса была добавлена в API.

Настройка сохранения оригинального IP-адреса клиента. Нужно сконфигурировать в спецификации сервиса два поля.

- ❑ `service.spec.externalTrafficPolicy`. Это поле определяет, к какой конечной точке сервис должен направлять внешний трафик: локальной, на уровне узла, или глобальной, на уровне кластера (используется по умолчанию). Второй вариант хорошо балансирует нагрузку, но не раскрывает исходный IP-адрес клиента и может добавить переход к другому узлу. Вариант с локальной конечной точкой сохраняет клиентский адрес и не добавляет лишних переходов (если тип сервиса равен `LoadBalancer` или `NodePort`). Его недостаток — плохое распределение нагрузки.
- ❑ `service.spec.healthCheckNodePort`. Это необязательное поле. Оно определяет номер порта, с помощью которого проверяется работоспособность сервиса. По умолчанию используется порт, выделенный узлу. Применяется к сервисам типа `LoadBalancer`, у которых поле `externalTrafficPolicy` равно `Local`.

Далее приведен пример:

```

{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "example-service"
  },
  "spec": {
    "ports": [{
      "port": 8765,
      "targetPort": 9376
    }],
    "selector": {
      "app": "example"
    },
    "type": "LoadBalancer"
    "externalTrafficPolicy": "Local"
  }
}

```

Потенциал внешнего равномерного распределения нагрузки

Внешние балансировщики распределяют нагрузку на уровне узла, хотя трафик направляется к конкретным подам. Представьте, что ваш сервис состоит из четырех подов, три из которых находятся на узле А, а один — на узле Б. В этом случае внешний балансировщик, скорее всего, равномерно распределит нагрузку между узлами. Таким образом три экземпляра подов примут на себя половину запросов (1/6 каждый), а вторая половина будет отведена единственному поду на узле Б. В будущем подобную проблему можно будет решить добавлением весов.

Балансирование нагрузки с помощью внутреннего сервиса

Балансирование нагрузки на уровне сервиса предназначено для распределения трафика внутри кластера Kubernetes без участия внешнего балансировщика. Это делается с помощью сервиса типа `clusterIP`. Внутренний балансировщик можно реализовать в виде сервиса типа `NodePort` и предоставлять прямой доступ к нему через заранее выделенный порт. Однако ввиду того, что это нестандартный сценарий использования, полезные возможности — разрыв SSL-соединения и экширование HTTP — будут недоступны.

На рис. 10.12 показано, как внутренний сервис (облако в центре) может играть роль балансировщика нагрузки и направлять трафик к одному из своих подов (конечно же, с помощью меток).

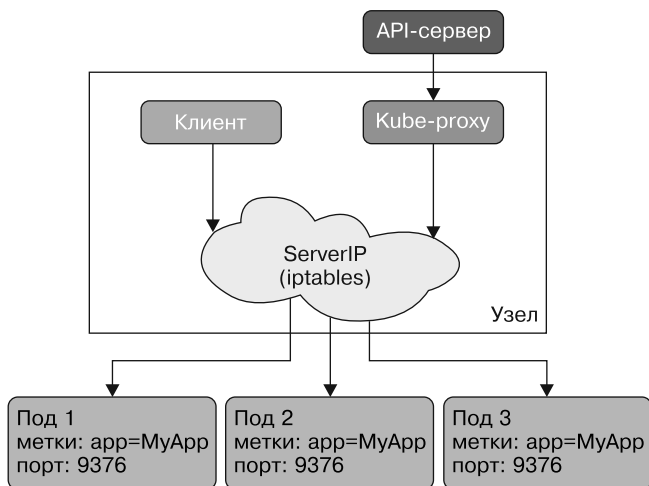


Рис. 10.12

Ingress

Ingress входит в состав Kubernetes и по большому счету представляет собой набор правил, которые позволяют направлять входящие соединения к сервисам кластера. Кроме того, некоторые контроллеры Ingress поддерживают следующие возможности:

- ❑ алгоритмы соединения;
- ❑ лимиты на запросы;
- ❑ перенаправление и переопределение URL-адресов;
- ❑ балансирование нагрузки на уровне TCP/UDP;
- ❑ разрыв SSL-соединений;
- ❑ контроль доступа и авторизацию.

Ingress описывается с помощью одноименного ресурса и обслуживается соответствующим контроллером. Важно отметить, что этот механизм все еще находится на стадии бета и пока не позволяет реализовать все необходимые возможности. Далее приведен пример ресурса Ingress, который направляет трафик к двум сервисам. Правила привязывают URL-адреса `foo.bar.com/foo` и `http://foo.bar.com/bar`, доступные извне, к сервисам `s1` и `s2` соответственно:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - path: /foo
            backend:
              serviceName: s1
              servicePort: 80
          - path: /bar
            backend:
              serviceName: s2
              servicePort: 80
```

Существует два официальных контроллера Ingress. Один из них, L7, предназначен только для GCE, второй более универсальный и позволяет конфигурировать NGINX через ConfigMap. Ingress-контроллер NGINX очень сложный и обладает множеством возможностей, которые все еще недоступны напрямую из ресурса Ingress. Он использует конечные точки API, чтобы направлять трафик непосредственно к подам, а также поддерживает Minikube, GCE, AWS, Azure и аппаратные кластеры. Его подробный обзор представлен на странице github.com/kubernetes/ingress-nginx.

HAProxy

Итак, мы познакомились с двумя способами балансирования нагрузки: внешним, через облачный сервис типа `LoadBalancer`, и внутренним, через сервис кластера типа `ClusterIP`. Теперь можем создать собственный провайдер для внешнего балансировщика, применяя `LoadBalancer` или еще один тип, `NodePort`. *HAProxy* (*High Availability Proxy* — высокодоступный прокси) — это зрелое проверенное опытом решение для внешнего распределения нагрузки. Оно считается лучшим выбором для локальных кластеров. Его можно интегрировать несколькими способами.

- ❑ Использовать `NodePort` и тщательно следить за выделением портов.
- ❑ Реализовать собственный интерфейс для провайдера балансировщика нагрузки.
- ❑ Запустить HAProxy изнутри в качестве единственной цели для ваших клиентских серверов на границе кластера (неважно, с балансированием или без).

HAProxy поддерживает все эти способы, однако рекомендуемый подход — с применением объектов `Ingress`. Сообщество разработало проект `service-loadbalancer`, реализующий решение для балансирования нагрузки на основе HAProxy. Он находится по адресу github.com/kubernetes/contrib/tree/master/service-loadbalancer.

Использование NodePort

Каждому сервису выделяется отдельный порт из заранее определенного диапазона. Последний обычно содержит порты с высокими значениями (от 30 000), чтобы избежать конфликтов с приложениями, применяющими стандартные (низкие) порты. В данном случае сервер HAProxy запускается за пределами кластера и знает о том, какой порт принадлежит тому или иному сервису. Благодаря этому он может перенаправлять трафик к любому узлу и Kubernetes через внутренний сервис, а балансировщик нагрузки направит его к подходящему поду (двойное балансирование). Конечно, это не идеальное решение, так как оно добавляет лишний переход. Чтобы обойти этот недостаток, можно обращаться к `Endpoints API` и хранить динамический список экземпляров подов для каждого сервиса. Это позволит направлять трафик непосредственно к подам.

Создание собственного провайдера для балансирования нагрузки на основе HAProxy

Этот подход чуть сложнее, но его преимуществами являются лучшая интеграция с Kubernetes и упрощенная миграция между локальными и облачными кластерами.

Запуск HAProxy внутри кластера Kubernetes

Этот подход подразумевает использование внутреннего балансировщика нагрузки на основе HAProxy. Последний может выполняться на нескольких узлах с общей конфигурацией для распределения входящих запросов между серверами кластера (на схеме, приведенной на рис. 10.13, показаны серверы Apache).

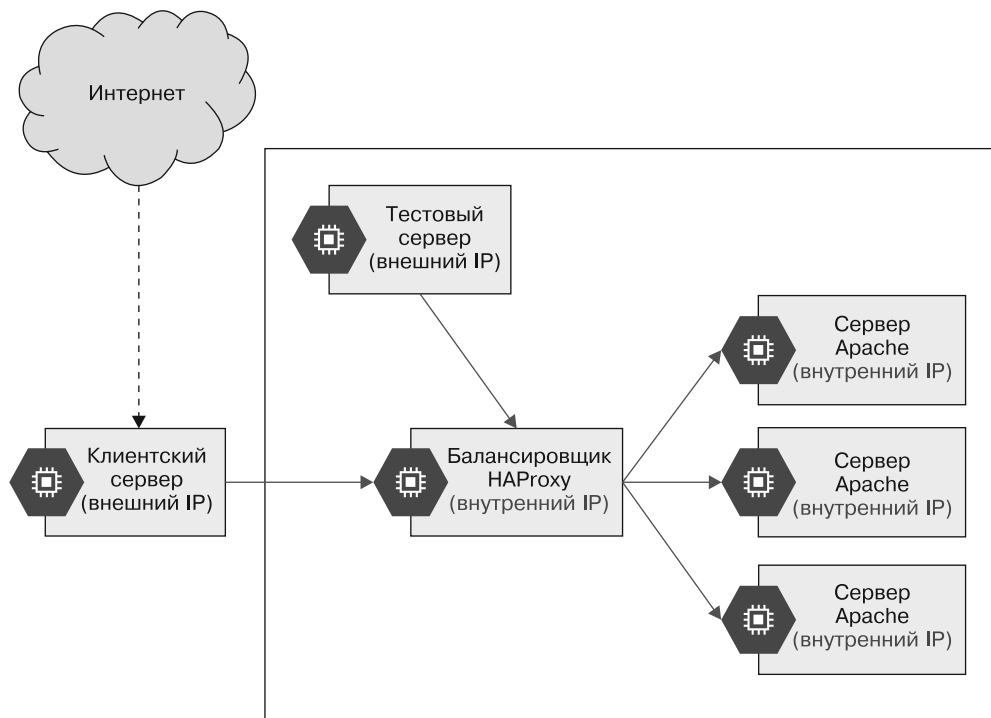


Рис. 10.13

Keepalived VIP

Keepalived VIP (от virtual IP — виртуальный IP-адрес) не имеет прямого отношения к распределению нагрузки. Это решение можно использовать совместно с Ingress-контроллером NGINX или сервисом LoadBalancer, основанным на HAProxy. Основной причиной его создания был тот факт, что поды, включая балансировщик (-и) нагрузки, могут перемещаться по кластеру Kubernetes. Это создает проблемы для клиентов за пределами сети, которым нужны стабильные точки входа. DNS-сервер часто не подходит из-за низкой производительности. Keepalived предоставляет высокопроизводительный виртуальный IP-адрес, который можно назначить Ingress-контроллеру NGINX или балансировщику нагрузки.

HAProxy. Эта система задействует базовые сетевые механизмы Linux, такие как IPVS (виртуальный сервер IP), и обеспечивает высокую доступность с помощью протокола VRRP (Virtual Redundancy Router Protocol — протокол резервного виртуального маршрутизатора). Все работает на четвертом сетевом уровне (TCP/UDP). Для конфигурации этого решения требуются определенные усилия и внимание к деталям. К счастью, команда Kubernetes создала проект `contrib`, который может послужить хорошей отправной точкой: <https://github.com/kubernetes/contrib/tree/master/keepalived-vip>.

Træfic

Træfic — это современный обратный HTTP-прокси и балансировщик нагрузки, который изначально разрабатывался для поддержки микросервисов. Он работает со многими платформами, включая Kubernetes, и умеет подстраивать свою конфигурацию автоматически и динамически. Это совершенно новая технология по сравнению с традиционными балансировщиками нагрузки. Список ее характеристик впечатляет:

- ❑ высокая скорость;
- ❑ единственный исполняемый файл (Go);
- ❑ крохотный официальный образ для Docker;
- ❑ Rest API;
- ❑ обновление конфигурации на лету без перезапуска процесса;
- ❑ предохранители, повторение попытки;
- ❑ Round Robin, распределение балансировщиков нагрузки;
- ❑ мониторинг (Rest, Prometheus, Datadog, Statsd, InfluxDB);
- ❑ изящный веб-интерфейс на основе AngularJS;
- ❑ поддержка WebSocket, HTTP/2, GRPC;
- ❑ журналы доступа (JSON, CLF);
- ❑ поддержка Let's Encrypt (HTTPS с автоматическим продлением);
- ❑ высокая доступность в режиме кластера.

Написание собственного дополнения для CNI

В этом разделе вы узнаете, что требуется для написания собственного CNI-дополнения. Сначала мы рассмотрим простейшую реализацию типа `loopback`. Затем изучим структуру дополнения и реализуем большую часть шаблона, пред-

назначенного для написания CNI-дополнения. В конце будет приведен пример дополнения-моста. Но прежде, чем приступить, вспомним, что представляет собой дополнение для CNI.

- ❑ CNI-дополнение — это исполняемый файл.
- ❑ Оно отвечает за подключение к сети новых контейнеров, назначение им IP-адресов и управление маршрутизацией.
- ❑ Контейнер — это сетевое пространство имен (в Kubernetes в качестве CNI-контейнера выступает под).
- ❑ Сетевые определения хранятся в формате JSON, но передаются дополнению через стандартный ввод (дополнение не читает никаких файлов).
- ❑ Дополнительную информацию можно предоставить через переменные окружения.

Знакомство с дополнением loopback

Наше дополнение просто добавляет интерфейс loopback. Оно настолько простое, что ему не нужна никакая конфигурация с данными о сети. Большинство CNI-дополнений реализованы на языке Golang, и loopback не исключение. Полный исходный код доступен по адресу github.com/containernetworking/plugins/blob/master/plugins/main/loopback.

Сначала рассмотрим раздел импорта. Здесь находится множество пакетов из проекта Containernetworking, размещенного на GitHub; они предоставляют много компонентов, необходимых для построения CNI-дополнения. Здесь также имеется пакет `netlink`, который добавляет/удаляет сетевые интерфейсы и назначает IP-адреса и маршруты. К пакету `skel` мы вернемся чуть позже:

```
package main
import (
    "github.com/containernetworking/cni/pkg/ns"
    "github.com/containernetworking/cni/pkg/skel"
    "github.com/containernetworking/cni/pkg/types/current"
    "github.com/containernetworking/cni/pkg/version"
    "github.com/vishvananda/netlink"
)
```

Затем дополнение реализует две команды, `cmdAdd` и `cmdDel`, которые вызываются при добавлении контейнера в сеть или его удалении из нее соответственно. Команда `cmdAdd` выглядит так:

```
func cmdAdd(args *skel.CmdArgs) error {
    args.IfName = "lo"
    err := ns.WithNetNSPath(args.Netns, func(_ ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
        if err != nil {
```

```

    return err // не проверяется
}

err = netlink.LinkSetUp(link)
if err != nil {
    return err // не проверяется
}

return nil
})
if err != nil {
    return err // не проверяется
}

result := current.Result{}
return result.Print()
}

```

Основная часть этой функции присваивает интерфейсу название `lo` (от `loopback`) и добавляет ссылку на сетевое пространство имен контейнера. Команда `del` делает противоположное:

```

func cmdDel(args *skel.CmdArgs) error {
    args.IfName = "lo"
    err := ns.WithNetNSPath(args.Netns, func(ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
        if err != nil {
            return err // не проверяется
        }

        err = netlink.LinkSetDown(link)
        if err != nil {
            return err // не проверяется
        }

        return nil
    })
    if err != nil {
        return err // не проверяется
    }
    result := current.Result{}
    return result.Print()
}

```

Функция `main` просто вызывает пакет `skel`, передавая ему командные функции. Пакет `skel` берет на себя запуск исполняемого файла CNI-дополнения и своевременный вызов функций `addCmd` и `delCmd`:

```

func main() {
    skel.PluginMain(cmdAdd, cmdDel, version.All)
}

```

Сборка CNI-дополнения на основе готового каркаса

Изучим пакет `skel` и посмотрим, что у него внутри. Начиная с точки входа `PluginMain()` он выполняет вызов `PluginMainWithError()`, перехватывает и направляет ошибки в стандартный вывод, а затем завершает работу:

```
func PluginMain(cmdAdd, cmdDel func(_ *CmdArgs) error, versionInfo
version.PluginInfo) {
    if e := PluginMainWithError(cmdAdd, cmdDel, versionInfo); e != nil {
        if err := e.Print(); err != nil {
            log.Print("Error writing error JSON to stdout: ", err)
        }
        os.Exit(1)
    }
}
```

Функция `PluginErrorWithMain()` создает экземпляр диспетчера, назначает ему все потоки ввода/вывода и переменные окружения, после чего вызывает из него метод `PluginMain()`:

```
func PluginMainWithError(cmdAdd, cmdDel func(_ *CmdArgs) error, versionInfo
version.PluginInfo) *types.Error {
    return ( dispatcher{
        Getenv: os.Getenv,
        Stdin: os.Stdin,
        Stdout: os.Stdout,
        Stderr: os.Stderr,
    }).pluginMain(cmdAdd, cmdDel, versionInfo)
}
```

Наконец, мы добрались до основной логики нашего каркаса. Функция `pluginMain()` берет аргументы `cmd` из окружения (включая конфигурацию, подаваемую на стандартный ввод), определяет, какая команда была вызвана, и вызывает подходящую функцию `plugin` (`cmdAdd` или `cmdDel`). Она также может вернуть информацию о версии:

```
func (t *dispatcher) pluginMain(cmdAdd, cmdDel func(_ *CmdArgs) error,
versionInfo version.PluginInfo) *types.Error {
    cmd, cmdArgs, err := t.getCmdArgsFromEnv()
    if err != nil {
        return createTypedError(err.Error())
    }

    switch cmd {
    case "ADD":
        err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdAdd)
    case "DEL":
        err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdDel)
    case "VERSION":
        err = versionInfo.Encode(t.Stdout)
    }
```

```

default:
    return createTypedError("unknown CNI_COMMAND: %v", cmd)
}

if err != nil {
    if e, ok := err.(*types.Error); ok {
        // Error не заворачивается внутрь Error
        return e
    }
    return createTypedError(err.Error())
}
return nil
}

```

Обзор дополнения-моста

Рассмотрим некоторые ключевые аспекты реализации дополнения-моста. Полный исходный код доступен по адресу github.com/containernetworking/plugins/blob/master/plugins/main/bridge.

Здесь определяется структура для конфигурации сети со следующими полями:

```

type NetConf struct {
    types.NetConf
    BrName          string 'json:"bridge"'
    IsGW            bool  'json:"isGateway"'
    IsDefaultGW     bool  'json:"isDefaultGateway"'
    ForceAddress    bool  'json:"forceAddress"'
    IPMasq         bool  'json:"ipMasq"'
    MTU            int   'json:"mtu"'
    HairpinMode     bool  'json:"hairpinMode"'
    PromiscMode     bool  'json:"promiscMode"'
}

```

Из-за ограниченного размера главы мы не станем разбирать назначение всех параметров и то, как они взаимодействуют между собой. Цель этого раздела — понять процесс движения данных и получить отправную точку для реализации собственного CNI-дополнения. Конфигурация загружается из JSON через функцию `loadNetConf()`. Это происходит в начале функций `cmdAdd()` и `cmdDel()`:

```
n, cniVersion, err := loadNetConf(args.StdinData)
```

Далее показана основная часть функции `cmdAdd()`. Она берет сетевую конфигурацию, поднимает интерфейс `veth`, добавляет подходящие IP-адреса с помощью дополнения IPAM и возвращает результаты:

```

hostInterface, containerInterface, err := setupVeth(netns, br, args.IfName,
n.MTU,
n.HairpinMode)
if err != nil {

```

```

    return err
}

// Вызываем дополнение IPAM и получаем конфигурацию,
// которую нужно применить
r, err := ipam.ExecAdd(n.IPAM.Type, args.StdinData)
if err != nil {
    return err
}

// Переводим вывод IPAM в текущий тип Result
result, err := current.NewResultFromResult(r)
if err != nil {
    return err
}

if len(result.IPs) == 0 {
    return errors.New("IPAM returned missing IP config")
}
result.Interfaces = []*current.Interface{brInterface, hostInterface,
    containerInterface}

```

Это лишь часть реализации. За кулисами остались назначение маршрутов и выделение IP-адресов. Полный исходный код довольно объемный. Можете исследовать его самостоятельно, чтобы получить полноценное представление о происходящем.

Резюме

В этой главе мы рассмотрели много материала. Организация сети — обширная область, в которой сочетаются разные оборудование, программное обеспечение, среды выполнения и навыки пользователей. Разработка комплексного сетевого решения, которое было бы устойчивым, безопасным, производительным и простым в обслуживании, — крайне сложная задача. В кластерах Kubernetes этим занимаются в основном облачные провайдеры. Но если вам нужно локальное или нестандартное решение, в вашем распоряжении огромное количество вариантов. Платформа Kubernetes очень гибкая и расширяемая. Это особенно относится к сети. Основные темы, которые мы обсудили, касались сетевой модели Kubernetes (плоское адресное пространство, в котором возможно взаимодействие между разными подами и всеми контейнерами внутри одного пода), поиска и обнаружения, сетевых дополнений Kubernetes, различных сетевых решений на разных уровнях абстракции (множество интересных вариаций), эффективного применения сетевых политик для управления трафиком внутри кластера и обзора решений для балансирования нагрузки. В конце главы вы познакомились с устройством реального CNI-дополнения.

Если вы не специалист в данной области, такой объем новой информации может показаться ошеломительным. Тем не менее у вас должно сложиться ясное представление о внутреннем устройстве сети в Kubernetes и о том, из каких взаимосвязанных элементов состоит полноценное сетевое решение. Вы также должны уметь проектировать собственные решения, идя на компромиссы, уместные в вашем конкретном случае.

В главе 11 мы двинемся дальше и научимся запускать Kubernetes в нескольких кластерах, на облачных платформах и в режиме Federation. Это важная часть географически распределенного развертывания и обеспечения предельной масштабируемости. Федеративные кластеры Kubernetes позволяют избавиться от локальных ограничений, но создают целый ряд новых проблем.

11

Запуск Kubernetes в нескольких облаках и многокластерном режиме

В этой главе мы выйдем на новый уровень и научимся запускать Kubernetes в нескольких облаках и в многокластерном режиме. Кластер Kubernetes — тесно интегрированная система, компоненты которой работают в относительной близости друг к другу и связаны между собой быстрой сетью (внутри реального вычислительного центра или зоны доступности облачного провайдера). Это хорошее решение для многих задач, но существует несколько важных сценариев, в которых масштабирование должно выходить за пределы одного кластера. Kubernetes Federation — это хорошо продуманный метод объединения нескольких кластеров в одну сущность. Мы рассмотрим следующие темы.

- ❑ Подробное знакомство с многокластерным режимом.
- ❑ Подготовка, настройка и администрирование Kubernetes Federation.
- ❑ Выполнение федеративных рабочих задач на нескольких кластерах.

Многокластерный режим

Концепция многокластерного (федеративного) режима довольно проста. Несколько кластеров Kubernetes объединяются в один логический кластер. Благодаря федеративному управляющему уровню клиенты видят систему как единое целое.

Общий принцип работы Kubernetes Federation показан на рис. 11.1.

Федеративный управляющий уровень состоит из федеративного API-сервера и федеративного диспетчера контроллеров. Первый направляет запросы ко всем кластерам, которые входят в федерацию, а второй берет на себя управление контроллерами всех ее членов, распределяя запросы с учетом изменений в каждом из них. На практике это довольно нетривиальная система, которую нельзя полностью инкапсулировать. Взаимодействие между подами и передача данных могут привести к огромным задержкам и денежным расходам. Сначала посмотрим, для чего нужен многокластерный режим, и попытаемся понять принцип работы федеративных

компонентов и ресурсов. После этого можно будет перейти к сложным аспектам — географической принадлежности, межкластерному планированию и федеративному доступу к данным.

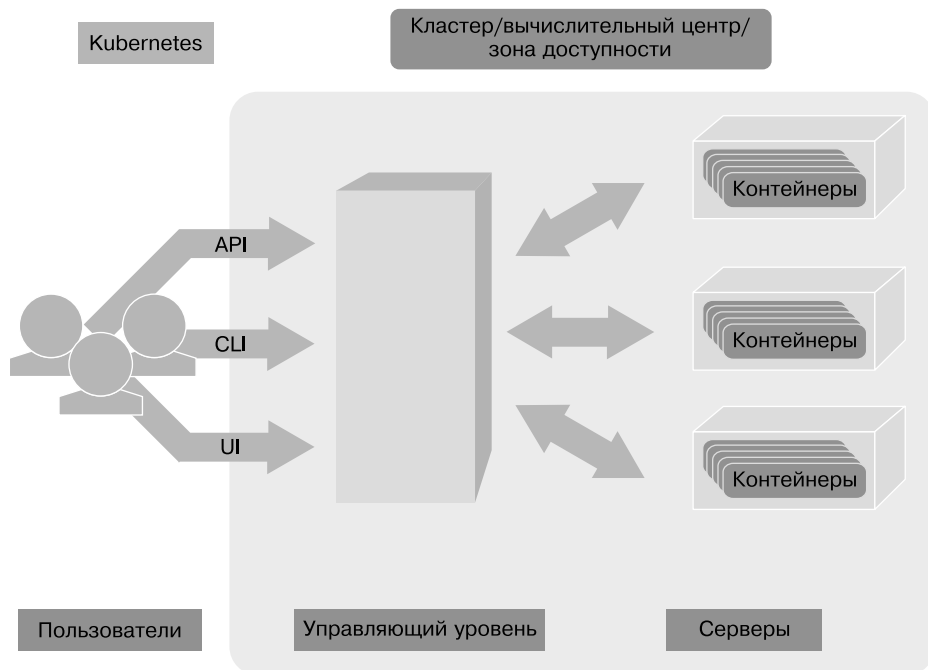


Рис. 11.1

Важные сценарии использования многокластерного режима

Существует четыре общих случая, когда имеет смысл применять многокластерный режим. Рассмотрим их.

Нехватка ресурсов

Публичные облачные платформы AWS, GCE и Azure — это отличный выбор, так как они обеспечивают множество преимуществ, но их нельзя назвать дешевыми. Поэтому многие крупные организации сделали капитальные вложения в собственные вычислительные центры. Некоторые компании сотрудничают с частными поставщиками вроде OVS, Rackspace или Digital Ocean. Если у вас есть ресурсы для содержания собственной инфраструктуры, кластер Kubernetes будет дешевле разместить локально, чем в облаке. Но что, если ваша рабочая нагрузка колеблется, требуя значительного, но кратковременного повышения мощности?

Допустим, нагрузка на вашу систему особенно сильно возрастает по выходным или в праздники. Традиционным решением было бы просто повысить мощность. Но во многих динамических ситуациях это довольно сложно. Вы можете выполнять основную часть работы в локальном вычислительном центре или на частной платформе, компенсируя нехватку ресурсов за счет дополнительного кластера Kubernetes, запущенного в каком-нибудь крупном облаке. Большую часть времени дополнительный кластер выключен (с остановкой серверов), но при необходимости вы сможете динамически повысить мощность своей системы, запустив некоторые из остановленных узлов. Kubernetes Federation делает такую конфигурацию относительно простой и позволяет избавиться от головной боли, связанной с планированием мощности и расходами на оборудование, которое в основном простаивает.

Такой подход называют *облачным ускорением*.

Конфиденциальные данные

Это почти полная противоположность нехватке ресурсов. Возможно, вы в восторге от облачных решений и вся ваша система работает в облаке, однако некоторая ее часть имеет дело с конфиденциальными данными. Правовые нормы или политика безопасности организации способны предписывать, что определенные информация и рабочие процессы должны находиться в полностью подконтрольной вам среде и могут подвергаться внешним проверкам. Гарантия того, что информация никогда не просочится из частного кластера Kubernetes в облако, очень важна. В то же время желательно обеспечивать доступность облачного кластера и возможность запуска в нем процессов, не связанных с конфиденциальными данными. Если рабочие процессы способны становиться конфиденциальными посреди выполнения, то требуются разработка и реализация надлежащей политики. Например, вы можете запретить переключение в конфиденциальный режим или перенести из облачного кластера рабочий процесс, который внезапно стал конфиденциальным. Еще один важный аспект — соблюдение законодательных норм разных стран, согласно которым некоторые данные должны оставаться в определенной географической области (обычно внутри страны). В таких случаях необходимо создать отдельный кластер в соответствующем регионе.

Привязка к поставщику услуг

Многие большие организации предпочитают иметь возможность выбирать среди нескольких поставщиков услуг. Привязка к какой-то одной платформе часто слишком рискованна, так как она способна попросту исчезнуть или не справиться с предъявляемыми требованиями. К тому же наличие нескольких поставщиков может стать хорошим аргументом при обсуждении цен. Платформа Kubernetes спроектирована таким образом, чтобы ее можно было запускать в разных облаках, частных хостингах и локальных вычислительных центрах.

Однако это непростой аспект. Если вы хотите быть уверены в том, что сможете быстро поменять провайдеров или перенести рабочую нагрузку с одного

провайдера на другого, то ваша система должна изначально работать сразу на нескольких платформах. Сделайте это самостоятельно или воспользуйтесь услугами компаний, которые помогают распределить Kubernetes между разными провайдерами. Поскольку разные платформы размещены в разных вычислительных центрах, вы автоматически получаете резервные копии и защиту от перебоев в работе того или иного поставщика.

Высокая доступность с географическим распределением

Высокая доступность подразумевает, что система продолжит предоставлять услуги, даже если некоторые ее части перестанут работать. В случае Kubernetes Federation отказать может целый кластер, обычно это происходит из-за перебоев в вычислительном центре или более глобальных проблем с провайдером платформы. Гарантией высокой доступности является избыточность. Геораспределенная избыточность означает наличие нескольких кластеров, размещенных в разных местах. Это могут быть разные зоны доступности, регионы и даже облачные провайдеры (см. предыдущий раздел). Работа в многокластерном режиме с избыточностью сопряжена с множеством проблем, которые нельзя игнорировать. Некоторые из них обсудим позже. Если решить все технические и организационные вопросы, высокая доступность позволит переключить трафик с отказавшего кластера на какой-то другой. Для пользователей этот процесс должен быть прозрачным, но лишь до определенной степени (во время переключения вероятны задержки и потеря/отказ текущих запросов или задач). Иногда системным администраторам приходится принимать дополнительные меры для выполнения переключения и разбираться с причинами поломки исходного кластера.

Федеративный управляющий уровень

Федеративный управляющий уровень состоит из двух компонентов, сочетание которых позволяет кластерам Kubernetes работать как единое целое.

Федеративный API-сервер

Федеративный API-сервер управляет кластерами Kubernetes, которые составляют федерацию. Состояние федерации, как и любого отдельного кластера, хранится в etcd, хотя, в сущности, оно представляет собой лишь список объединенных кластеров. Каждый член федерации хранит свое состояние в собственном экземпляре etcd. Основные задачи федеративного API-сервера — взаимодействие с федеративным диспетчером контроллеров и направление запросов к разным кластерам. Члены федерации могут не знать, что они входят в ее состав, — они продолжают работать как прежде.

На рис. 11.2 показаны отношения между федеративными API-сервером и диспетчером контроллеров, а также кластерами Kubernetes, входящими в федерацию.

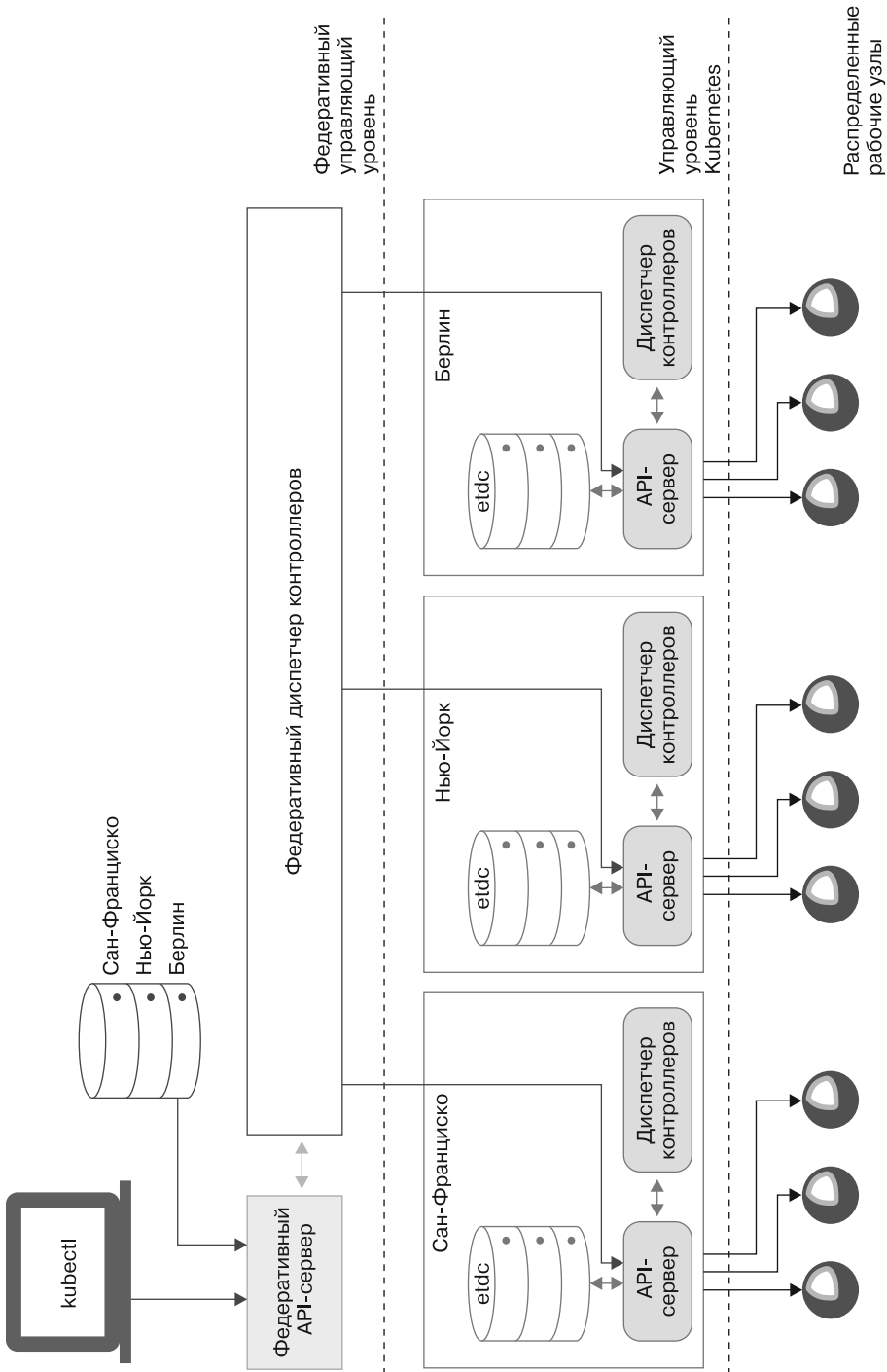


Рис. 11.2

Федеративный диспетчер контроллеров

Федеративный диспетчер контроллеров следит за тем, чтобы реальное состояние федерации совпадало с желаемым. Все необходимые изменения он направляет соответствующим кластерам. Его двоичный файл содержит разные контроллеры для всех федеративных ресурсов, которые мы рассмотрим позже в данной главе. Хотя здесь используется похожая логика управления: диспетчер отслеживает изменения и в случае необходимости приводит кластер в нужное состояние. Это делается для каждого члена федерации.

На рис. 11.3 продемонстрирован нескончаемый цикл управления.

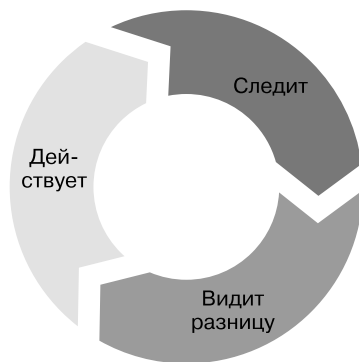


Рис. 11.3

Федеративные ресурсы

Kubernetes Federation все еще находится на ранней стадии развития. В Kubernetes 1.10 лишь некоторые ресурсы можно сделать федеративными. Мы рассмотрим их в этом разделе. Для создания федеративного ресурса утилите `kubectl` передается аргумент командной строки `--context=federation-cluster`. В таком случае команда направляется федеративному API-серверу, который автоматически рассылает ее по всем своим кластерам.

Федеративная карта ConfigMap

Федеративные карты ConfigMap очень полезны. Они помогают централизовать конфигурацию приложений, которые могут быть размазаны по разным кластерам.

Создание федеративной карты ConfigMap

Вот пример создания федеративной карты ConfigMap:

```
> kubectl --context=federation-cluster create -f configmap.yaml
```

Как видите, единственное отличие от создания ConfigMap в однокластерном режиме состоит в наличии контекста. Федеративная карта ConfigMap помещается

в базу данных `etcd` на управляющем уровне, но ее копии хранятся также в каждом члене федерации. Таким образом, каждый кластер может оставаться независимым и не требует доступа к управляющему уровню.

Просмотр федеративной карты ConfigMap

Вы можете просмотреть ConfigMap, обратившись к управляющему уровню или члену федерации. Во втором случае необходимо указать контекст, то есть имя кластера:

```
> kubectl --context=cluster-1 get configmap configmap.yaml1
```

Обновление федеративной карты ConfigMap

Важно отметить, что при создании через управляющий уровень каждый кластер получит идентичный экземпляр ConfigMap. Но поскольку та же карта хранится еще и в управляющем кластере, мы лишаемся единого источника истины. Позже ConfigMap можно будет модифицировать отдельно для каждого члена федерации (хотя это и не рекомендуется), в результате чего мы получим неоднородную конфигурацию. В некоторых случаях это имеет смысл, но в таких ситуациях я советую настраивать каждый кластер напрямую. Создавая федеративную карту ConfigMap, вы тем самым заявляете, что она должна быть общей для всех кластеров. Чтобы обновить ее сразу для всех членов федерации, используйте параметр `--context=federation-cluster`.

Удаление федеративной карты ConfigMap

Как вы, наверное, уже догадались, удаление происходит как обычно, только с указанием контекста:

```
> kubectl --context=federation-cluster delete configmap
```

Есть лишь один небольшой нюанс. На момент выхода Kubernetes 1.10 удаление федеративной карты ConfigMap не затрагивает остальные экземпляры ConfigMap, автоматически созданные на каждом кластере. Вам придется удалять их отдельно. Таким образом, если федерация состоит из трех кластеров: `cluster-1`, `cluster-2` и `cluster-3`, для полного удаления ConfigMap нужно выполнить три дополнительные команды:

```
> kubectl --context=cluster-1 delete configmap
> kubectl --context=cluster-2 delete configmap
> kubectl --context=cluster-3 delete configmap
```

В дальнейшем эта неувязка будет устранена.

Федеративный контроллер DaemonSet

Федеративный контроллер DaemonSet практически ничем не отличается от обычного. Его создание и работа с ним производятся через управляющий уровень (заданием параметра `--context=federation-cluster`), который передает все изменения

членам федерации. В результате вы можете быть уверены в том, что экземпляр DaemonSet запущен на всех узлах каждого кластера.

Федеративное развертывание

Ресурсы федеративного развертывания развиты чуть лучше. Все реплики по умолчанию равномерно распределяются между кластерами. Если у вас три кластера, то при федеративном развертывании 15 подов каждый кластер получит по пять реплик. По аналогии с другими федеративными ресурсами управляющий уровень сохранит федеративное развертывание с 15 репликами, а затем выполнит три развертывания (по одному на кластер) с пятью репликами в каждом. Регулировать количество реплик в кластерах можно с помощью аннотации `federation.kubernetes.io/deployment-preferences`. На момент выхода Kubernetes 1.10 федеративное развертывание все еще находится в состоянии альфа, но в будущем вместо аннотации в его конфигурационном файле появится полноценное поле.

Федеративные события

Федеративные события отличаются от других федеративных ресурсов. Они хранятся исключительно на управляющем уровне и не передаются членам федерации. Их можно запрашивать как обычно, с помощью параметра `--context=federation-cluster`:

```
> kubectl --context=federation-cluster get events
```

Федеративное горизонтальное масштабирование подов

Экспериментальная поддержка федеративного *горизонтального масштабирования подов* (horizontal pod scaling или HPA) появилась лишь недавно, в Kubernetes 1.9. Чтобы ее включить, при запуске API-сервера необходимо установить следующий флаг:

```
--runtime-config=api/all=true
```

Это важная функция, поскольку одно из основных преимуществ многокластерного режима — динамическое распределение рабочей нагрузки между разными кластерами без вмешательства со стороны. Федеративное HPA-масштабирование использует HPA-контроллеры отдельных членов федерации. Оно равномерно распределяет нагрузку между кластерами, основываясь на том, какое минимальное и максимальное количество реплик мы запросили. В будущем пользователи смогут описывать более сложные политики для HPA.

Возьмем, к примеру, федерацию с четырьмя кластерами. Мы хотим, чтобы в ней всегда работало не меньше шести, но не больше 16 подов. Для этого достаточно определить такой манифест:


```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: cool-app
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: cool-app
  minReplicas: 6
  maxReplicas: 16
  targetCPUUtilizationPercentage: 80

```

Чтобы инициировать федеративное НРА-масштабирование, используйте следующую команду:

```
> kubectl --context=federation-cluster create federated-hpa.yaml
```

Что в итоге произойдет? Федеративный управляющий уровень создаст стандартные НРА-контроллеры в каждом из четырех кластеров с количеством реплик в диапазоне от двух до четырех. Это самый экономный вариант, удовлетворяющий требованиям Kubernetes Federation. Давайте подумаем, почему. Если в каждом кластере может быть не больше четырех реплик, то общее их количество не превысит $4 \cdot 4 = 16$, что соответствует нашим критериям. Минимальное количество реплик не должно быть меньше двух, благодаря чему их общее количество не опустится ниже $4 \cdot 2 = 8$ (напомним, нам нужно как минимум шесть реплик). Даже при отсутствии какой-либо нагрузки на систему у нас будет не меньше восьми реплик, хотя хватило бы и шести. С этим нельзя ничего поделать, так как распределение происходит равномерно между всеми кластерами. Если указать для кластерных НРА-контроллеров поле `minReplicas=1`, общее количество реплик в кластере может опуститься до $4 \cdot 1 = 4$, а это уже меньше федеративного минимума (шесть). В будущем пользователи смогут задавать более сложные схемы распределения.

Можно использовать кластерные селекторы (введены в Kubernetes 1.7), чтобы ограничить федеративный объект до подмножества членов. Итак, если мы хотим как минимум шесть и максимум 15, можно равномерно распределить его среди трех кластеров вместо четырех, и каждый кластер будет иметь минимум два и максимум пять.

Федеративный доступ

Федеративный доступ делает больше, чем просто создает соответствующие входные объекты в каждом кластере. Одна из основных особенностей федеративного доступа заключается в том, что, если весь кластер падает, он может направлять трафик в другие кластеры. Начиная с Kubernetes 1.4, федеративный доступ поддерживается на Google Cloud Platform как на GKE, так и на GCE. В будущем добавят поддержку гибридного облака для федеративного доступа.

Федеративный доступ выполняет следующие задачи:

- ❑ создает объекты Kubernetes, входящие в каждый член кластерной федерации;
- ❑ обеспечивает единый логический балансировщик нагрузки L7 с одним IP-адресом для всех объектов доступа к кластеру;
- ❑ отслеживает работоспособность и емкость служебных подов, находящихся за входящими объектами в каждом кластере;
- ❑ обязательно перенаправляет клиентские соединения на конечную точку работоспособного сервиса при различных сбоях, таких как сбой модуля, кластера, зоны доступности или всей области, если в федерации есть один работающий кластер.

Создание федеративного доступа

Создайте федеративный доступ, обращаясь к федеративному управляющему уровню:

```
> kubectl --context=federation-cluster create -f ingress.yaml
```

Федеративный управляющий уровень создаст соответствующий вход в каждом кластере. Все кластеры будут использовать одно и то же пространство имен и имя для объекта Ingress:

```
> kubectl --context=cluster-1 get ingress myingress
```

NAME	HOSTS	ADDRESS	PORTS	AGE
ingress	*	157.231.15.33	80, 443	1m

Запрос маршрутизации с федеративным доступом

Контроллер федеративного доступа направляет запросы в ближайший кластер. Объекты Ingress предоставляют один или несколько IP-адресов (с помощью поля `Status.Loadbalancer.Ingress`), которые остаются статическими на время своего существования. Когда внутренний или внешний клиент подключается к IP-адресу объекта входа, специфичного для кластера, он будет перенаправлен в один из контейнеров в этом кластере. Но когда клиент подключается к IP-адресу федеративного входного объекта, он автоматически маршрутизируется через самый короткий путь сети к работоспособному контейнеру в ближайшем к источнику запроса кластере. Так, например, запросы HTTP(S) от интернет-пользователей в Европе будут направляться непосредственно в ближайший кластер в Европе, имеющий подходящую мощность. Если в Европе таких кластеров нет, запрос будет перенаправлен на следующий ближайший кластер (часто в США).

Обработка сбоев с федеративным доступом

Существует две категории отказов:

- ❑ сбой пода;
- ❑ сбой кластера.

Поды могут сбоить по многим причинам. В правильно сконфигурированном кластере Kubernetes (неважно, член федерации он или нет) управляющие модули будут управляться сервисами и ReplicaSets, которые способны автоматически обрабатывать отказы подов. Он не должен влиять на кросс-кластерную маршрутизацию и балансировку нагрузки, выполняемую федеративным доступом. Кластер может выйти из строя из-за проблем с центром обработки данных или глобальной связью. В этом случае федеративные сервисы и федеративные ReplicaSets гарантируют, что другие кластеры в федерации будут работать с достаточным количеством подов для поддержки рабочей нагрузки, а федеративный доступ станет заботиться о маршрутизации запросов клиентов от отказавшего кластера. Чтобы воспользоваться возможностью автоматического исцеления, клиенты всегда должны подключаться к объекту федеративного доступа, а не к отдельным членам кластера.

Федеративные задачи

Федеративные задачи действуют аналогично внутрикластерным задачам. Федеративный управляющий уровень создает задания в базовых кластерах и равномерно разделяет нагрузку с учетом параллелизма задач и отслеживания их завершения. Например, если федерация имеет четыре кластера и вы создаете спецификацию федеративного задания с параллелизмом, равным восьми, и завершением, равным 24, то в каждом кластере будет создано задание с параллелизмом два и завершением шесть.

Федеративное пространство имен

Пространства имен Kubernetes используются в кластере для изоляции независимых областей и поддержки развертывания с несколькими арендаторами. Федеративные пространства имен предоставляют одинаковые возможности для кластерной федерации. API идентичен. Когда клиент обращается к федеративному управляющему уровню, он получает доступ только к пространствам имен, которые запросил, и имеет право доступа ко всем кластерам в федерации.

Вы применяете те же команды и добавляете `--context=federation-cluster`:

```
> kubectl --context=federation-cluster create -f namespace.yaml
> kubectl --context=cluster-1 get namespaces namespace
> kubectl --context=federation-cluster create -f namespace.yaml
```

Федеративные объекты ReplicaSet

Лучше всего использовать развертывания и федеративные развертывания для управления репликами в кластере или федерации. Но если по какой-то причине вы предпочитаете работать непосредственно с ReplicaSets, Kubernetes поддерживает федеративные объекты ReplicaSet. Федеративного контроллера репликации не существует, потому что ReplicaSets заменяет собой контроллеры репликации.

Когда вы создаете федеративные ReplicaSets, задание управляющего уровня должно гарантировать, что количество реплик по всему кластеру соответствует федеративной конфигурации ReplicaSets. Управляющий уровень создаст обычный ReplicaSet в каждом члене федерации. Все кластеры по умолчанию будут иметь равное (или как можно более близкое) количество реплик, чтобы сумма доходила до указанного количества реплик.

Вы можете контролировать количество реплик для каждого кластера с помощью аннотации `federation.kubernetes.io/replica-set-preferences`.

Соответствующая структура данных выглядит следующим образом:

```
type FederatedReplicaSetPreferences struct {
    Rebalance bool
    Clusters map[string]ClusterReplicaSetPreferences
}
```

Если `Rebalance` равно `true`, рабочие реплики при необходимости могут перемещаться между кластерами. Карта кластеров определяет настройки ReplicaSets для каждого кластера. Если в качестве ключа задано `*`, то все неуказанные кластеры будут использовать этот набор предпочтений. Если нет записи `*`, то реплики будут выполняться только в кластерах, которые отображаются на карте. Кластеры, принадлежащие федерации, но не указанные на карте, не будут иметь запланированных подов (для этого шаблона пода).

Индивидуальные настройки ReplicaSets для каждого кластера задаются с применением следующей структуры данных:

```
type ClusterReplicaSetPreferences struct {
    MinReplicas int64
    MaxReplicas *int64
    Weight int64
}
```

По умолчанию `MinReplicas` равен 0. По умолчанию `MaxReplicas` не ограничен. Вес равен 0.

Федеративные секреты

Федеративные секреты просты. Когда вы создаете федеративный секрет как обычно, через управляющий уровень, он распространяется на весь кластер. Вот и все.

Сложности

Пока федерация кажется не особенно сложной. Вы группируете множество кластеров, обращаетесь к ним через управляющий уровень, и все просто реплицируется на все кластеры. Но есть тяжелые и сложные факторы и основные концепции, которые усложняют эту упрощенную схему. Богатые возможности Kubernetes в основном обусловлены способностью этой системы инкапсулировать в себе

множество функций. В одном кластере, развернутом полностью в одном физическом центре обработки данных или в зоне доступности, где все компоненты подключены к быстрой сети, Kubernetes очень эффективна сама по себе. В кластерной федерации Kubernetes ситуация другая. Задержки, затраты на передачу данных и перемещение контейнеров между кластерами приводят к разным компромиссам. В зависимости от варианта использования создание федерации может потребовать от разработчиков и операторов системы дополнительного внимания, планирования и обслуживания. Кроме того, некоторые из федеративных ресурсов не так зрелы, как их локальные коллеги, и это добавляет неопределенности.

Федеративная рабочая единица

Рабочая единица в кластере Kubernetes — под. Вы не можете разбить ее на более мелкие части. Весь контейнер всегда разворачивается вместе и будет иметь один и тот же жизненный цикл. Должен ли под оставаться единицей работы для кластерной федерации? Возможно, было бы разумнее использовать более крупную единицу работы, такую как ReplicaSet, развертывание или сервис, с определенным кластером. Если кластер выходит из строя, весь ReplicaSet, развертывание или сервис переходят в другой кластер. А как насчет коллекции тесно связанных ReplicaSets? Ответы на эти вопросы не всегда быстро находятся и могут даже динамически меняться по мере развития системы.

Близкое местоположение

О приоритете местоположения следует серьезно позаботиться. Когда контейнеры могут быть распределены между кластерами? Каковы отношения между этими подами? Существуют ли какие-то требования к близости между подами или между подами и другими ресурсами, такими как хранилище? Есть несколько основных категорий:

- ☐ сильно связанные;
- ☐ слабо связанные;
- ☐ предпочтительно связанные;
- ☐ строго развязанные;
- ☐ равномерно распространенные.

При проектировании системы, планировании и распределении услуг и подов по всей федерации важно следить за тем, чтобы требования к близости местоположения всегда соблюдались.

Сильная связанность

Требование сильной связанности применяется к приложениям, в которых контейнеры должны находиться в одном кластере. Если вы разбиваете контейнеры, приложение будет сбоить (вероятно, из-за требований в реальном времени, которые

не могут быть удовлетворены при создании сетей через кластеры), а стоимость может оказаться слишком высокой (поды могут иметь доступ к множеству локальных данных). Единственный способ переместить такие тесно связанные приложения в другой кластер — запустить полную копию, включая данные, в другом кластере, а затем закрыть приложение в действующем кластере. Если объем данных слишком велик, приложение может оказаться практически не перемещаемым и чувствительным к катастрофическому сбою. Это самая сложная ситуация, с которой приходится иметь дело, и по возможности вы должны разработать свою систему, чтобы избежать строго связанных требований.

Слабая связанность

Слабо связанные приложения лучше всего задействовать, когда рабочая нагрузка мешает параллельности и каждый под не должен знать о других подах или получать доступ к большому количеству данных. В этих ситуациях контейнеры могут быть запланированы для кластеров только на основе емкости и использования ресурсов по всей федерации. При необходимости поды способны без проблем перемещаться из одного кластера в другой. Таков, например, сервис проверки состояния без сохранения, который выполняет определенные расчеты и получает все свои данные в самом запросе и не запрашивает и не записывает данные всей федерации. Он просто проверяет свой ввод и возвращает вызывающему вердикт: действителен или недействителен.

Предпочтительная связанность

Приложения с предпочтительной связанностью работают лучше, когда все поды находятся в одном кластере или когда поды и данные расположены рядом. Однако это требование не жесткое. Например, можно работать с приложениями, которые требуют только конечной согласованности, когда какое-либо приложение для всей федерации периодически синхронизирует состояние приложения во всех кластерах. В этих случаях распределение выполняется явно в отношении одного кластера, но для страховки остается возможность для запуска или миграции в другие кластеры в стрессовой ситуации.

Строгая связанность

Некоторые сервисы имеют защиту от сбоев или высокие требования к доступности, из-за которых происходит разделение между кластерами. Нет смысла запускать три реплики критического сервиса, если все они могут быть запланированы в один и тот же кластер, потому что этот кластер просто становится *единой точкой отказа* (SPOF).

Равномерное распределение

Равномерное распространение — это когда экземпляр сервиса, ReplicaSet или под должны запускаться в каждом кластере. Этот подход напоминает работу DaemonSet, но экземпляры распределяются не по узлам, а по кластерам. Хороший

пример — кэш Redis, за которым стоит некое внешнее постоянное хранилище. Поды в каждом кластере должны иметь собственный кластерный кэш Redis, чтобы избежать доступа к центральному хранилищу, что способно замедлить процесс или стать узким местом. В то же время нет необходимости в более чем одном сервисе Redis для каждого кластера (он может быть распределен по нескольким подам в одном кластере).

Кросс-кластерное планирование

Кросс-кластерное планирование идет рука об руку с привязкой местоположения. Когда создается новый блок или отсутствует нужный под и необходимо запланировать замену, куда следует идти? Текущая кластерная федерация не обрабатывает все сценарии и параметры близости местоположения, о которых мы говорили ранее. На данный момент кластерная федерация обрабатывает слабо связанные (включая взвешенное распределение) и строго связанные (с достаточной степенью уверенности в том, что количество реплик соответствует количеству кластеров). Все остальное потребует, чтобы вы не использовали кластерную федерацию. Вам нужно будет добавить собственный пользовательский уровень федерации, который учитывает специализированные проблемы и может вместить более сложные варианты планирования.

Федеративный доступ к данным

Это сложная проблема. Если у вас много данных и подов, работающих в нескольких кластерах (вероятно, на разных континентах), и нужно быстро получить к ним доступ, придется выбирать один из нескольких неприятных вариантов.

- ☐ Репликация данных в каждый кластер (медленная репликация, дорогостоящая передача, дорогое хранение и сложность синхронизации и устранения ошибок).
- ☐ Удаленный доступ к данным (медленный доступ, затратный доступ и, может быть, SPOF).
- ☐ Создание сложного гибридного решения с покластерным кэшированием самых свежих данных (сложные/устаревшие данные, и все равно нужно передать много данных).

Федеративное автомасштабирование

В настоящее время федеративное автомасштабирование не поддерживается. Можно использовать два измерения масштабирования, а также их комбинацию:

- ☐ масштабирование по кластеру;
- ☐ добавление кластеров в федерацию и удаление из нее;
- ☐ гибридный подход.

Рассмотрим относительно простой сценарий слабо связанного приложения, работающего в трех кластерах с пятью подами в каждом из них. В какой-то момент

15 подов перестают справляться с нагрузкой. Нужно расширить их возможности. Мы способны увеличить количество подов на кластер, но если сделаем это на уровне федерации, то получим по шесть подов в каждом кластере. Мы увеличили емкость федерации тремя подами, когда требуется только один. Конечно, если кластеров больше, проблема возрастает. Другой вариант — выбрать кластер и изменить его емкость. Это допустимо с аннотациями, но теперь мы будем явно управлять емкостью по всей федерации. Работа может очень быстро усложниться, если имеется много кластеров, в которых работают сотни сервисов с динамически меняющимися требованиями.

Добавление нового кластера еще сложнее. Куда именно его следует добавить? Здесь нет необходимости в дополнительной доступности, поэтому данный аспект не играет роли в принятии решения. Все сводится к лишней емкости. Вдобавок создание нового кластера часто требует сложной первоначальной настройки, и может потребоваться несколько дней, чтобы утвердить различные квоты на общедоступных облачных платформах. Гибридный подход увеличивает емкость существующих кластеров в федерации до достижения некоторого порога, а затем начнет добавлять кластеры. Преимущество такого подхода заключается в следующем: когда близок предел емкости для каждого кластера, вы начинаете готовить новые кластеры, которые приступят к работе, когда будет необходимо. Кроме того, это требует больших усилий, и вы платите в большей степени за гибкость и масштабируемость.

Управление многокластерным режимом Kubernetes

Управление многокластерным режимом Kubernetes включает в себя множество действий, которые не распространяются на работу с одиночным кластером. Существует два способа настройки федерации. Затем вам нужно рассмотреть возможность каскадного удаления ресурсов, балансировку нагрузки между кластерами, защиту от сбоев кластеров, федеративное обнаружение сервисов. Давайте рассмотрим все это подробно.

Настройка многокластерного режима с нуля

Примечание: этот подход устарел, вместо него используется Kubefed. Я опишу его здесь для читателей, применяющих более старые версии Kubernetes.

Чтобы создать кластерную федерацию Kubernetes, нужно запустить следующие компоненты управляющего уровня:

```
etcd  
federation-apiserver  
federation-controller-manager
```

Один из самых простых способов сделать это — задействовать образ Hyperkube «все в одном»: github.com/kubernetes/kubernetes/tree/master/cluster/images/hyperkube.

Федеративный API-сервер и федеративный диспетчер контроллеров могут выполняться как поды в существующем кластере Kubernetes, но, как обсуждалось ранее, для обеспечения отказоустойчивости и высокой доступности лучше использовать их в отдельном кластере.

Начальная настройка

В первую очередь нужно запустить Docker и и загрузить версию Kubernetes со скриптами, которые мы будем применять далее. Текущий выпуск 1.5.3. Загрузите последнюю версию:

```
> curl -L
https://github.com/kubernetes/kubernetes/releases/download/v1.5.3/
kubernetes.tar.gz | tar xvzf -
> cd kubernetes
```

Теперь нужно создать каталог для файлов конфигурации федерации и установить в него переменную среды `FEDERATION_OUTPUT_ROOT`. Для того чтобы легко было выполнять очистку, лучше всего создать новый каталог:

```
> export FEDERATION_OUTPUT_ROOT="${PWD}/output/federation"
> mkdir -p "${FEDERATION_OUTPUT_ROOT}"
```

Теперь можно инициализировать федерацию:

```
> federation/deploy/deploy.sh init
```

Использование официального образа Hyperkube

В рамках любого релиза Kubernetes официальные образы выпуска помещаются в `gcr.io/google_containers`. Чтобы использовать образы в этом репозитории, сделайте так, чтобы поля образа контейнера в файлах конфигурации в `${FEDERATION_OUTPUT_ROOT}` указывали на образ `gcr.io/google_containers/hyperkube`, который включает в себя двоичные файлы `federation-apiserver` и `federation-controller-manager`.

Запуск федеративного управляющего уровня

Мы готовы развернуть федеративный управляющий уровень, выполнив следующую команду:

```
> federation/deploy/deploy.sh deploy_federation
```

Команда запустит компоненты управляющего уровня в виде подов и создаст сервис типа `LoadBalancer` для федеративного API-сервера и запрос постоянного тома, подкрепленный динамическим постоянным томом для `etcd`.

Чтобы убедиться, что все правильно создано в пространстве имен федерации, введите:

```
> kubectl get deployments --namespace=federation
```

Вы должны увидеть следующее:

NAME	DESIRED	CURRENT	UP-TO-DATE
federation-controller-manager	1	1	1
federation-apiserver	1	1	1

Также можете проверить новые записи в своем файле `kubeconfig`, используя представление конфигурации `kubectl`. Обратите внимание на то, что сейчас динамическое выделение работает только для AWS и GCE.

Регистрация кластеров Kubernetes в федерации

Для регистрации кластера в федерации нам нужен секрет, чтобы поговорить с ним. Создадим секрет в кластере Kubernetes. Предположим, что конфигурация `kubeconfig` целевого кластера находится в каталоге `/cluster-1/kubeconfig`. Запустите следующую команду, чтобы создать `secret`:

```
> kubectl create secret generic cluster-1 --namespace=federation
--from-file=/cluster-1/kubeconfig
```

Конфигурация для кластера выглядит так:

```
apiVersion: federation/v1beta1
kind: Cluster
metadata:
  name: cluster1
spec:
  serverAddressByClientCIDRs:
  - clientCIDR: <cidr-клиента>
    serverAddress: <адрес-апи-сервера>
  secretRef:
    name: <имя-секрета>
```

Нужно установить параметры `<cidr-клиента>`, `<адрес-апи-сервера>` и `<имя-секрета>`. `<имя-секрета>` — это название только что созданного секрета. `serverAddressByClientCIDRs` содержит различные адреса серверов, которые клиенты могут использовать в соответствии с их CIDR. Мы можем установить публичный IP-адрес сервера с CIDR `0.0.0.0/0`, который будет соответствовать всем клиентам. Кроме того, если вы хотите, чтобы внутренние клиенты применяли серверный `clusterIP`, установите его как `serverAddress`. Клиент CIDR в этом случае будет соответствовать только IP-адресам подов, запущенных в данном кластере.

Зарегистрируем кластер:

```
> kubectl create -f /cluster-1/cluster.yaml --context=federation-cluster
```

Посмотрим, правильно ли он зарегистрирован:

```
> kubectl get clusters --context=federation-cluster
NAME STATUS VERSION AGE
cluster-1 Ready 1m
```

Обновление KubeDNS

Кластер зарегистрирован в федерации. Пришло время обновить `kube-dns`, чтобы кластер мог маршрутизировать запросы федеративного сервиса. Начиная с Kubernetes 1.5 или более поздней версии это делается передачей флага `-federations` в `kube-dns` через конфигурацию `kube-dns ConfigMap`:

```
--federations=${ИМЯ_ФЕДЕРАЦИИ}=${ИМЯ_DNS_ДОМЕНА}
```

Содержимое файла `ConfigMap` выглядит так:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  federations: <federation-name>=<federation-domain-name>
```

Замените параметры `ИМЯ_ФЕДЕРАЦИИ` и `ИМЯ_DNS_ДОМЕНА` собственными значениями.

Отключение многокластерного режима

Если вы хотите отключить многокластерный режим, просто выполните следующую команду:

```
federation/deploy/deploy.sh destroy_federation
```

Настройка многокластерного режима с помощью Kubefed

В Kubernetes 1.5 появился новый экспериментальный инструмент командной строки Alpha под названием `Kubefed`, который поможет вам управлять своими федеративными кластерами. Задача `Kubefed` заключается в том, чтобы упростить развертывание нового федеративного управляющего уровня в кластере Kubernetes и добавлять или удалять кластеры из существующего федеративного управляющего уровня. Начиная с Kubernetes 1.6, этот инструмент находится на стадии бета.

Получение Kubefed

До версии Kubernetes 1.9 `Kubefed` был частью клиентских двоичных файлов Kubernetes. Вы получите `kubectl` и `Kubefed`. Далее представлены инструкции по загрузке и установке в Linux:

```
curl -LO
https://storage.googleapis.com/kubernetes-release/release/${RELEASE-VERSION}/
kubernetes-client-linux-amd64.tar.gztar -xzf kubernetes-client-linuxamd64.tar.gz
```

```
sudo cp kubernetes/client/bin/kubefed /usr/local/bin
sudo chmod +x /usr/local/bin/kubefed
sudo cp kubernetes/client/bin/kubect1 /usr/local/bin
sudo chmod +x /usr/local/bin/kubect1
```

Внесите необходимые изменения, если используете другую ОС или хотите установить другую версию. Начиная с Kubernetes 1.9, Kubefed доступен в отдельном федеративном репозитории:

```
curl -LO
https://storage.cloud.google.com/kubernetes-federation-release/
release/${RELEASE-VERSION}/federation-client-linux-amd64.tar.gztar -xzf
federationclient-linux-amd64.tar.gz
sudo cp federation/client/bin/kubefed /usr/local/bin
sudo chmod +x /usr/local/bin/kubefed
```

Вы можете установить Kubectl отдельно, следуя инструкциям, которые найдете по адресу kubernetes.io/docs/tasks/tools/install-kubectl/.

Выбор хост-кластера

Федеративный управляющий уровень может представлять собой отдельный выделенный кластер или входить в состав существующего кластера. Какой вариант лучше, решать вам. В кластере размещаются компоненты, составляющие федеративный управляющий уровень. Убедитесь, что у вас есть запись `kubeconfig` в локальной конфигурации `kubeconfig`, которая соответствует вашему кластеру.

Чтобы убедиться, что нужная запись `kubeconfig` существует, введите следующее:

```
> kubectl config get-contexts
```

Вы должны увидеть что-то вроде этого:

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
	cluster-1	cluster-1		

Имя контекста, `cluster-1`, будет предоставлено позже при развертывании федеративного управляющего уровня.

Развертывание федеративного управляющего уровня

Пришло время начать использовать Kubefed. Команде `kubefed init` требуются три аргумента:

- ☐ название федерации;
- ☐ контекст хост-кластера;
- ☐ суффикс доменного имени для ваших федеративных услуг.

Следующая команда, например, разворачивает федеративный управляющий уровень с федерацией имен, контекст кластера хоста `cluster-1`, также действителен DNS-провайдер `coredns` (а еще `google-clouddns` и `aws-route53`), суффикс домена `kubernetes-ftw.com`:

```
> kubefed init federation --host-cluster-context=cluster-1 --dns-provider  
coredns --dns-zone-name="kubernetes-ftw.com"
```

Конечно, должен быть суффикс DNS для домена DNS, которым вы управляете.

`kubefed init` устанавливает федеративный управляющий уровень в кластере хоста и добавляет запись для федеративного API-сервера в локальной конфигурации `kubeconfig`. Из-за ошибки Kubernetes может не создавать пространство имен по умолчанию. В таком случае вам придется сделать это самостоятельно. Введите следующую команду:

```
> kubectl create namespace default --context=federation
```

Не забудьте установить текущий контекст в федерацию, чтобы `kubectl` нацелился на федеративный управляющий уровень:

```
> kubectl config use-context federation
```

Добавление кластера к федерации

Как только управляющий уровень будет успешно развернут, нужно добавить в федерацию кластеры Kubernetes. Для этого в `Kubefed` предусмотрена команда `join`. Команде `kubefed join` требуются следующие аргументы:

- ☐ имя кластера, который нужно добавить;
- ☐ контекст основного кластера.

Например, чтобы добавить в федерацию новый кластер с именем `cluster-2`, введите следующее:

```
kubefed join cluster-2 --host-cluster-context=cluster-1
```

Правила и настройки именования

Имя кластера, которое вы добавляете для `kubefed join`, должно быть допустимым по RFC 1035. RFC 1035 допускает только буквы, цифры и дефисы, а имя должно начинаться с буквы.

Более того, для федеративного управляющего уровня требуются учетные данные объединенных кластеров. Эти учетные данные получены из локальной конфигурации `kubeconfig`. Команда `kubefed join` использует имя кластера, указанное в качестве аргумента, для поиска контекста кластера в локальной конфигурации `kubeconfig`. Не находя соответствующего контекста, он завершается с ошибкой.

Это может вызвать проблемы в тех случаях, когда имена контекстов для каждого кластера в федерации не соответствуют правилам именования RFC 1035. В таких

случаях можете задать имя кластера, соответствующее правилам именования RFC 1035, и указать контекст кластера с помощью флага `--cluster-context`. Например, если контекст кластера, к которому вы присоединяетесь, является `cluster-3` (символ подчеркивания недопустим), выполните следующее:

```
kubefed join cluster-3 --host-cluster-context=cluster-1 --cluster-context=cluster-3
```

Секретное имя

Учетные данные кластера, которых требует федеративный управляющий уровень, как описано в предыдущем разделе, хранятся как секрет в основном кластере. Имя секрета также выводится из имени кластера.

Однако имя объекта `secret` в Kubernetes должно соответствовать спецификации имени поддомена DNS, описанной в RFC 1123. Если это не так, передайте `secret name` в `kubefed join`, используя флаг `--имя-секрета`. Например, если имя кластера — это `cluster-4`, а имя секрета — `4secret` (начинать с буквы нельзя), вы можете присоединиться к кластеру, выполнив следующее:

```
kubefed join cluster-4 --host-cluster-context=cluster-1 --secret-name=4secret
```

Команда `kubefed join` автоматически создаст секрет.

Исключение кластера из федерации

Чтобы исключить кластер из федерации, запустите команду `kubefed unjoin` с именем кластера и контекстом хост-кластера:

```
kubefed unjoin cluster-2 --host-cluster-context=cluster-1
```

Выключение многокластерного режима

Правильная очистка федеративного управляющего уровня в этой бета-версии Kubefed реализована не полностью. Однако сейчас удаление пространства имен системы федерации должно удалять все ресурсы, за исключением постоянного тома, динамически выделенного для `etcd` федеративного управляющего уровня. Вы можете удалить (`delete`) пространство имен федерации, выполнив следующую команду:

```
> kubectl delete ns federation-system
```

Каскадное удаление ресурсов

Кластерная федерация Kubernetes часто управляет федеративным объектом на управляющем уровне, а также соответствующими объектами в каждом кластере Kubernetes. Каскадное удаление федеративного объекта означает, что соответствующие объекты в кластерах, участвующих в Kubernetes, также будут удалены.

Это происходит не автоматически. По умолчанию удаляется только объект федеративного управляющего уровня. Чтобы активизировать каскадное удаление, необходимо установить следующий параметр:

```
DeleteOptions.orphanDependents=false
```

В Kubernetes 1.5 только следующие федеративные ресурсы поддерживают каскадное удаление:

- ☐ развертывание;
- ☐ контроллер DaemonSets;
- ☐ доступ;
- ☐ пространство имен;
- ☐ объекты ReplicaSets;
- ☐ секреты.

Другие объекты следует удалять вручную в каждом отдельном кластере. К счастью, начиная с Kubernetes 1.6 все федеративные объекты поддерживают каскадное удаление.

Балансировка нагрузки между несколькими кластерами

Динамическая балансировка нагрузки между кластерами — непростая задача, и лучше не возлагать ее на Kubernetes. Балансировка нагрузки будет выполняться за пределами кластерной федерации Kubernetes. Но с учетом динамического характера Kubernetes даже внешний балансировщик нагрузки должен будет собрать много информации о том, какие сервисы и поды работают в каждом кластере. Альтернативное решение — реализовать на федеративном управляющем уровне балансировщик нагрузки L7, который будет служить диспетчером трафика для всей федерации. В одном из более простых случаев каждый сервис выполняется в выделенном кластере и балансировщик нагрузки просто направляет в него весь трафик. В случае отказа кластера сервис переносится в другой кластер, и теперь балансировщик нагрузки направляет весь трафик в новый кластер. Это обеспечивает устойчивость решения к ошибкам и высокую доступность на уровне кластера.

Оптимальное решение сможет поддерживать федеративные сервисы и учитывать следующие дополнительные факторы:

- ☐ географическое расположение клиента;
- ☐ использование ресурсов каждого кластера;
- ☐ квоты ресурсов и автоматическое масштабирование.

На рис. 11.4 показано, как балансировщик нагрузки L7 на GCE направляет клиентские запросы к ближайшему кластеру.

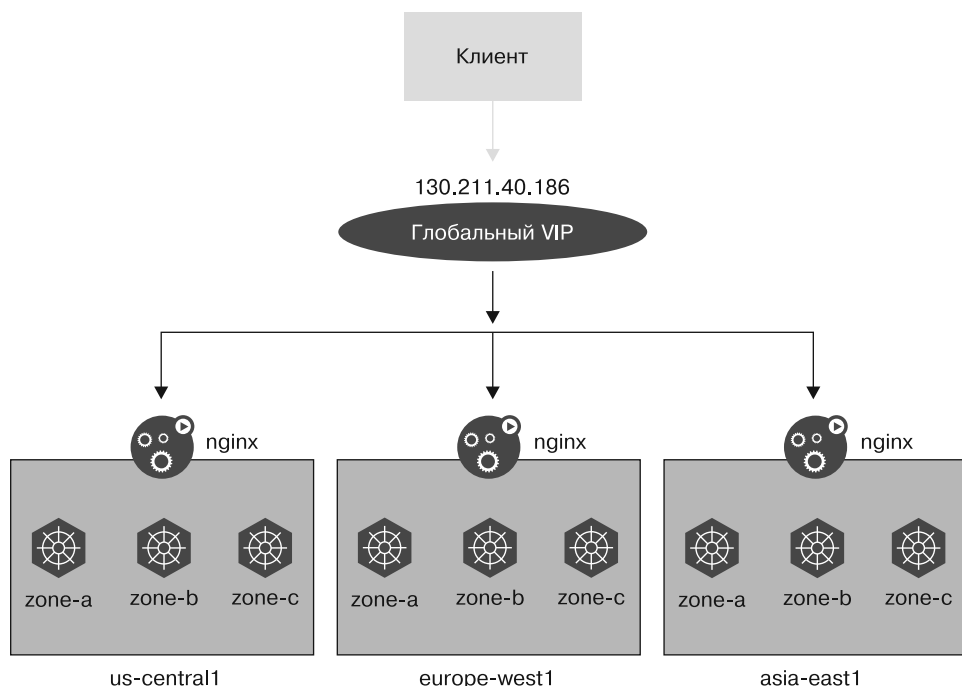


Рис. 11.4

Переключение на другие кластеры в случае сбоя

Федеративное переключение на другие кластеры — сложный процесс. Предположим, что кластер в федерации сбойт. Один из вариантов решения проблемы заключается в том, чтобы другие кластеры забирали себе его нагрузку. Теперь возникает вопрос: как распределить нагрузку на другие кластеры:

- ☐ равномерно;
- ☐ запустить новый кластер;
- ☐ выбрать наиболее близкий из существующих кластеров (возможно, в том же регионе)?

Каждое из этих решений тонко взаимодействует с федеративной балансировкой нагрузки, географическим распределением с высокой доступностью, управлением затратами в разных кластерах и безопасностью.

Теперь сбоивший кластер возвращается в сеть. Должен ли он постепенно снова взять на себя первоначальную рабочую нагрузку? Что, если он вернется, но с ограниченными способностями или упрощенной сетью? Существует много комбинаций режимов отказа, которые могут усложнить процесс восстановления.

Обнаружение федеративных услуг

Обнаружение федеративных сервисов тесно связано с балансировкой нагрузки в федерации. Прагматичная конфигурация включает в себя глобальный балансировщик нагрузки L7, который распределяет запросы к федеративным входным объектам в кластерах федерации.

Преимущество этого подхода заключается в том, что контроль остается за федерацией Kubernetes, которая со временем сможет работать с большим количеством типов кластеров (в настоящее время только AWS и GCE), а также учитывать эффективность использования кластеров и различные ограничения.

Альтернатива специализированному сервису поиска позволяет клиентам напрямую подключаться к сервисам в отдельных кластерах, но при этом теряются все перечисленные преимущества.

Федеративная миграция

Федеративная миграция связана с несколькими темами, которые мы уже рассмотрели, а именно близость местоположения, федеративное планирование и высокая доступность. По своей сути федеративная миграция означает перемещение целого приложения или некоторой его части из одного кластера в другой (и в более общем смысле из M кластеров в N кластеров). Миграция федерации может произойти в результате следующих событий.

- ❑ Событие низкой емкости в кластере или сбой кластера.
- ❑ Изменение политики планирования (мы больше не используем провайдера облака X).
- ❑ Изменение ценообразования на ресурсы (поставщик облачных услуг Y снизил цены, поэтому давайте перейдем туда).
- ❑ Новый кластер был добавлен в федерацию или удален из нее (выполняется перебалансировка подов приложения).

Строго связанные приложения могут быть легко перемещены частично или полностью по одному поду за раз в один или несколько кластеров (в рамках применимых ограничений политики, таких как `PrivateCloudOnly`).

Для приложений с преимущественной связью система федерации должна сначала найти один кластер с достаточной емкостью для размещения всего приложения, зарезервировать ее и постепенно перемещать приложение, один ресурс или более за раз, в новый кластер в пределах некоторого ограниченного периода времени (и, возможно, в пределах предопределенного окна обслуживания).

Строго связанные приложения (за исключением тех, которые нельзя переносить) требуют, чтобы федеративная система выполняла следующие действия.

- ❑ Запустить полное приложение реплики в целевом кластере.
- ❑ Скопировать постоянные данные в новый экземпляр приложения (возможно, перед запуском модулей).
- ❑ Переключить трафик пользователя.
- ❑ Закрыть оригинальный экземпляр приложения.

Выполнение федеративных рабочих нагрузок

Федеративные рабочие нагрузки — это рабочие нагрузки, которые обрабатываются одновременно в нескольких кластерах Kubernetes. Это относительно легко сделать для слабо связанных и распределенных приложений. Однако, если большая часть обработки может выполняться параллельно, часто в конце есть точка соединения или по крайней мере постоянное центральное хранилище, к которому следует обращаться на чтение и запись. Все усложняется, если несколько подов одного и того же сервиса должны взаимодействовать между кластерами или если сервисы (каждый из которых можно сделать федеративным) должны работать вместе и синхронизироваться для выполнения чего-либо.

Федерация Kubernetes поддерживает федеративные сервисы, которые обеспечивают отличную основу для таких федеративных рабочих нагрузок. Ключевыми особенностями федеративных сервисов являются обнаружение сервисов, балансировка нагрузки между кластерами и отказоустойчивость зоны доступности.

Создание федеративного сервиса

Федеративный сервис создает соответствующие сервисы в кластерах федерации. Например, для создания федеративного сервиса NGINX (при условии, что его конфигурация находится в файле `nginx.yaml`) введите следующее:

```
> kubectl --context=federation-cluster create -f nginx.yaml
```

Вы можете проверить, что сервис создан в каждом кластере (допустим, в `cluster-2`):

```
> kubectl --context=cluster-2 get services nginx
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx	10.63.250.98	104.199.136.89	80/TCP	9m

Все созданные сервисы во всех кластерах будут называться одинаково и входить в одно и то же пространство имен, что логично, поскольку они составляют один логический сервис.

Статус вашего федеративного сервиса автоматически отражает статус исходных сервисов Kubernetes в режиме реального времени:

```
> kubectl --context=federation-cluster describe services nginx
Name:                nginx
Namespace:           default
Labels:              run=nginx
Selector:            run=nginx
Type:               LoadBalancer
IP:
LoadBalancer Ingress: 105.127.286.190, 122.251.157.43, 114.196.14.218,
                      114.199.176.99, ...
Port:               http      80/TCP
Endpoints:          <none>
Session Affinity:   None
No events.
```

Добавление подов

В версии Kubernetes 1.10 нам по-прежнему нужно добавлять поды в каждый кластер членов федерации. Это делается с помощью команды `kubectl run`. В будущем выпуске федеративный API-сервер Kubernetes сможет делать это автоматически. Это сократит процесс на один шаг. Обратите внимание: когда вы используете команду запуска `kubectl`, Kubernetes автоматически добавляет метку `run` в под на основе имени образа. В следующем примере, который запускает под NGINX на пяти кластерах Kubernetes, имя изображения — `nginx` (с игнорированием версии), поэтому добавляется следующая метка:

```
run=nginx
```

Это необходимо, потому что сервис использует ее для идентификации своих подов. Если вы применяете другую метку, придется добавить ее явно:

```
for C in cluster-1
    cluster-2
    cluster-3
    cluster-4
    cluster-5
do
    kubectl --context=$C run nginx --image=nginx:1.11.1-alpine --port=80
done
```

Проверка общедоступных записей DNS

После того как предыдущие поды были успешно запущены и начали прослушивать соединения, Kubernetes сообщит о них как о работоспособных конечных точках сервиса в этом кластере (через автоматические проверки работоспособности). Кластерная федерация Kubernetes, в свою очередь, будет рассматривать каждый

из этих участков сервиса как работоспособный и задействовать их, автоматически настраивая соответствующие общедоступные записи DNS. Чтобы проверить это, используйте предпочтительный интерфейс от настроенного DNS-провайдера. Например, ваша федерация может быть настроена для работы с Google Cloud DNS и управляемым доменом DNS `example.com`:

```
> gcloud dns managed-zones describe example-dot-com
creationTime: '2017-03-08T18:18:39.229Z'
description: Example domain for Kubernetes Cluster Federation
dnsName: example.com.
id: '7228832181334259121'
kind: dns#managedZone
name: example-dot-com
nameServers:
- ns-cloud-a1.googledomains.com.
- ns-cloud-a2.googledomains.com.
- ns-cloud-a3.googledomains.com.
- ns-cloud-a4.googledomains.com.
```

Введите следующую команду, чтобы увидеть фактические записи DNS:

```
> gcloud dns record-sets list --zone example-dot-com
```

Если ваша федерация настроена на применение сервиса `aws route53 DNS`, используйте следующие команды:

```
> aws route53 list-hosted-zones
```

Затем введите:

```
> aws route53 list-resource-record-sets --hosted-zone-id K9PBV0X1QTOVBX
```

Разумеется, вы можете задействовать стандартные инструменты DNS, такие как `nslookup` или `dig`, чтобы проверить, что записи DNS обновлены правильно. Вероятно, придется немного подождать, пока ваши изменения не распространятся по сети. Как вариант, можете указать прямо на провайдера DNS:

```
> dig @ns-cloud-e1.googledomains.com ...
```

Я всегда предпочитаю наблюдать изменения DNS прямо после того, как они были правильно распространены, чтобы сообщить пользователям о готовности к работе.

Обнаружение федеративного сервиса

KubeDNS встроен в Kubernetes и является одним из его основных компонентов. KubeDNS использует локальный (`cluster-local`) DNS-сервер, а также соглашения об именовании для составления хорошо продуманных доменных имен (распределенных между областями видимости). Например, `the-service` разрешает `the-service` в пространстве имен по умолчанию `namespace`, тогда как

`the-service.the-namespace` разрешается сервисом `the-service` в пространстве имен `the-имя_пространства_имен`space — отдельном от значения по умолчанию `the-service`. Поды могут легко найти и получить доступ к внутренним услугам с помощью KubeDNS. Кластерная федерация Kubernetes расширяет механизм до нескольких кластеров. Основная концепция неизменна, но добавляется еще один уровень федерации. DNS-имя сервиса теперь состоит из `<имя_сервиса>.<имя_пространства_имен>.<имя_федерации>`. Таким образом, доступ к внутренним сервисам по-прежнему можно получать, применяя исходное соглашение об именах `<сервиса>.<имя_пространства_имен>`. Тем не менее клиенты, которые хотят получить доступ к федеративному сервису, используют федеративное имя, которое в итоге будет перенаправлено на один из кластеров — членов федерации для обработки запроса.

Это соглашение об именовании, одобренное федерацией, помогает также предотвратить попадание внутреннего кластерного трафика в другие кластеры по ошибке.

Используя предыдущий пример с сервисом NGINX и только что описанную форму DNS-имени федеративного сервиса, рассмотрим следующую ситуацию: модуль в кластере в зоне доступности `cluster-1` должен получить доступ к сервису NGINX. Вместо того чтобы задействовать традиционное локальное доменное имя сервиса `nginx.the-namespace`, которое автоматически расширяется до `nginx.thenamespace.svc.cluster.local`, теперь он может применять имя федеративного DNS-сервиса — `nginx.the-namespace.the-federation`. Если в локальном кластере существует работоспособный экземпляр, будет возвращен локальный (обычно `10.x.y.z`) IP-адрес этого сервиса (кластерно локальный KubeDNS). Это почти эквивалентно разрешению без федеративности услуг (почти, потому что KubeDNS фактически возвращает как `CNAME`, так и `A`-запись для локальных федеративных сервисов, но приложения не обращают внимания на эту незначительную техническую разницу).

Расширение DNS. Если сервис не существует в локальном кластере (или существует, но не имеет работоспособных подов), DNS-запрос автоматически расширяется, чтобы найти внешний IP-адрес, ближайший к зоне доступности запрашивающей стороны. KubeDNS выполняет это автоматически и возвращает соответствующий `CNAME`. В дальнейшем он будет разрешен как IP-адрес одного из поддерживающих подов сервиса.

Вам не нужно полагаться на автоматическое расширение DNS. Вы можете просто предоставить `CNAME` сервиса в конкретном кластере напрямую или в определенном регионе. Например, в GCE/GKE можно указать запись `nginx.the-namespace.svc.europe-west1.example.com`. Эта запись сошлется на резервный под в одном из кластеров в Европе (при условии, что там есть кластеры и работоспособные резервные поды).

Внешние клиенты не используют расширение DNS, но если они хотят нацелиться на некоторые ограниченные подмножества федерации, допустим, конкретный

регион, то могут предоставить полностью квалифицированный CNAME сервиса как пример. Поскольку эти имена обычно длинные и громоздкие, стоит добавить некоторые статистические записи CNAME:

```
eu.nginx.example.com      CNAME nginx.the-namespace.the-federation.svc.
                           europe-west1.example.com.
us.nginx.example.com      CNAME nginx.the-namespace.the-federation.svc.
                           us-central1.example.com.
nginx.example.com         CNAME nginx.the-namespace.the-federation.svc.
                           example.com.
```

Следующая схема (рис. 11.5) показывает, как федеративный поиск работает в нескольких кластерах.

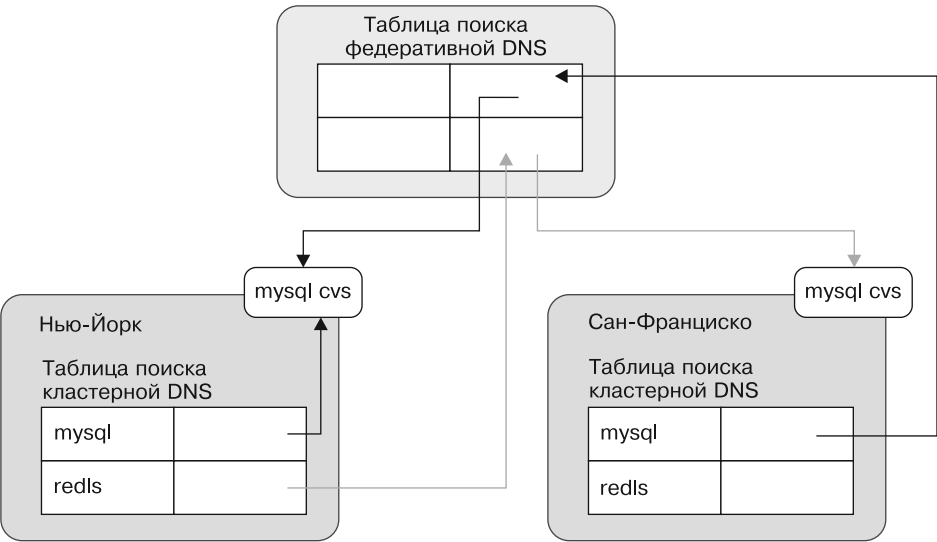


Рис. 11.5

Обработка отказов подов и целых кластеров

Работа подов, которые не отвечают на запросы, прекращается в течение нескольких секунд. Федеративный управляющий уровень контролирует работоспособность кластеров и конечных точек вашего федеративного сервиса в разных кластерах. Он будет вводить их в эксплуатацию и выводить из нее по мере необходимости, например, когда все конечные точки услуги, весь кластер или целая зона доступности ухудшатся. Собственная латентность кэширования DNS (для DNS Federated Service по умолчанию составляет 3 минуты) приведет к тому, что клиенты будут отправлены в альтернативный кластер в случае катастрофического сбоя. Однако,

учитывая количество дискретных IP-адресов, которые могут быть возвращены для каждой конечной точки регионального сервиса (см., например, us-central1, который имеет три альтернативы), многие клиенты, имея соответствующую конфигурацию, автоматически перейдут на один из альтернативных IP-адресов за меньшее время.

Решение проблем

Когда дела идут плохо, вам нужно определить проблему и пути ее решения. Разберем несколько общих проблем и способы их диагностики/устранения.

Не удалось подключиться к федеративному API-серверу

Обратитесь к следующему решению.

- ☐ Убедитесь, что федеративный API-сервер запущен.
- ☐ Убедитесь, что клиент (kubectl) настроен правильно, с соответствующими конечными точками API и учетными данными.

Федеративный сервис создается успешно, но в базовых кластерах не создается сервис

Решение таково.

- ☐ Убедитесь, что кластеры зарегистрированы в федерации.
- ☐ Убедитесь, что федеративный API-сервер смог подключиться и пройти аутентификацию во всех кластерах.
- ☐ Проверьте, что квоты достаточны.
- ☐ Проверьте журналы на наличие других проблем:

kubectl logs federation-controller-manager --namespace federation

Резюме

В этой главе мы рассмотрели важную тему — многокластерный режим Kubernetes. Этот режим все еще находится в стадии бета-тестирования и немного сыроват, но уже пригоден для использования. Пока развертываний не так уж много, и официально поддерживаемыми целевыми платформами сейчас являются AWS и GCE/GKE, но в пользу облачной федерации говорит многое. Это очень важный элемент для построения масштабируемых систем на Kubernetes.

Мы обсудили мотивы и варианты применения многокластерного режима Kubernetes, компоненты федеративного управляющего уровня и федеративные объекты Kubernetes. Кроме того, рассмотрели хуже поддерживаемые аспекты федерации — персонализированное планирование, интеграцию доступа к данным

и автоматическое масштабирование. Затем поговорили о том, как запускать несколько кластеров Kubernetes, включая настройку многокластерного режима Kubernetes, добавление кластеров в федерацию и их удаление из нее, а также балансировку нагрузки, федеративное противостояние сбоям, когда что-то пойдет не так, обнаружение сервиса и миграцию.

На этом этапе у вас должно сложиться четкое представление о текущем состоянии федерации, а также о том, что нужно для использования существующих возможностей, предоставляемых Kubernetes, и что вы должны будете реализовать самостоятельно, чтобы расширить неполные или незрелые функции. В зависимости от варианта применения придется определить, еще слишком рано или все же стоит сделать решительный шаг. Разработчики, работающие с многокластерным режимом Kubernetes, продвигаются быстро, поэтому очень вероятно, что к моменту, когда вам нужно будет принять решение, он будет гораздо более зрелым и проверенным в деле.

В следующей главе мы будем разбираться в содержимом Kubernetes и выяснять, как его настроить. Одним из основных архитектурных принципов платформы Kubernetes является то, что она доступна через полнофункциональный REST API. Инструмент командной строки `kubectl` построен поверх API Kubernetes и обеспечивает интерактивность для всего спектра Kubernetes. Однако программный доступ к API дает вам большую гибкость для расширения и дополнения Kubernetes. Существуют библиотеки клиентов на многих языках, которые позволяют вам использовать Kubernetes извне, — их можно интегрировать в существующие системы.

Kubernetes является модульной платформой. Многие основные параметры ее работы могут быть настроены и/или расширены. В частности, можете добавлять определяемые пользователем ресурсы, интегрировать их с объектной моделью Kubernetes и получать выгоду от услуг управления Kubernetes, хранения в `etcd`, взаимодействия через API и унифицированного доступа к встроенным и настраиваемым объектам.

Вы уже видели хорошо расширяемые компоненты, такие как сетевое взаимодействие и контроль доступа через дополнения CNI и пользовательские классы хранения. Однако Kubernetes идет еще дальше и позволяет настраивать сам планировщик, который контролирует назначение подов узлам.

12 Настройка Kubernetes: API и дополнения

В этой главе мы рассмотрим API Kubernetes и научимся работать с Kubernetes программным путем через прямой доступ к API, разберем клиент Python, а затем будем автоматизировать `kubectl`. Затем мы поговорим о расширении API Kubernetes с помощью настраиваемых ресурсов. В конце главы вы познакомитесь с различными дополнениями, поддерживаемыми Kubernetes. Многие аспекты работы Kubernetes — модульные и предназначены для расширения. Мы рассмотрим несколько типов дополнений, отвечающих за пользовательские планировщики, авторизацию, контроль доступа, пользовательские метрики и тома. Наконец, поговорим о расширении `kubectl` и добавлении ваших собственных команд.

Мы разберем следующие темы.

- ❑ Работа с API Kubernetes.
- ❑ Расширение API Kubernetes.
- ❑ Написание дополнений Kubernetes и `kubectl`.
- ❑ Написание веб-хуков.

Работа с API Kubernetes

API Kubernetes всеобъемлющий и охватывает всю функциональность Kubernetes. Как и следовало ожидать, он огромен. Но он хорошо разработан с использованием передового опыта и последователен. Если вы понимаете основные принципы, то сможете найти все, что вам нужно знать.

OpenAPI

OpenAPI позволяет поставщикам API определять свои операции и модели, а разработчикам — автоматизировать инструменты и генерировать клиенты для общения с этим API-сервером на своем любимом языке. Платформа Kubernetes некоторое время поддерживала *Swagger 1.2* (более старая версия спецификации

OpenAPI), но спецификация была неполной и неправильной, что затрудняло создание на ней инструментов или клиентов.

В Kubernetes 1.4 была добавлена экспериментальная поддержка спецификации OpenAPI (известной как Swagger 2.0, прежде чем она была передана инициативе OpenAPI) и обновлены текущие модели и операции. В Kubernetes 1.5 поддержка спецификации OpenAPI была оформлена окончательно путем автоматической генерации непосредственно из источника Kubernetes; это позволяет полностью синхронизировать спецификацию и документацию с будущими изменениями в операциях или моделях.

Новая спецификация позволяет улучшить документацию API и автоматически генерируемый клиент для Python, который мы рассмотрим позже.

Спецификация модульная и разделяется по групповой версии. Это на будущее. Вы можете запускать несколько серверов API, поддерживающих разные версии. Приложения могут постепенно перейти к более новым версиям.

Структура спецификации подробно объясняется в определении спецификации OpenAPI. Команда Kubernetes использовала рабочие теги для разделения всех версий группы и предоставления как можно большего количества информации о путях или операциях и моделях. Для конкретной операции документируются все параметры, методы вызова и ответы. Результат впечатляет.

Настройка прокси

Для упрощения доступа можно сконфигурировать прокси-сервер на основе Kubectl:

```
> kubectl proxy --port 8080
```

Теперь можно получить доступ к серверу API по адресу `http://localhost:8080`, и запрос достигнет того же API-сервера Kubernetes, для которого настроен kubectl.

Непосредственный доступ к API Kubernetes

API Kubernetes можно легко исследовать. Просто просмотрев URL-адрес API-сервера `http://localhost:8080`, вы получите прекрасный JSON-документ, который описывает все доступные операции (см. ключ `paths`).

Чтобы сэкономить место, я привожу лишь неполный список:

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
```

```

"/apis/apps",
"/apis/storage.k8s.io/v1",
.
.
.
"/healthz",
"/healthz/ping",
"/logs",
"/metrics",
"/swaggerapi/",
"/ui/",
"/version"
]
}

```

Вы можете обратиться к любому пути. Например, вот ответ от конечной точки `/api/v1/namespaces/default`:

```

{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "creationTimestamp": "2017-12-25T10:04:26Z",
    "name": "default",
    "resourceVersion": "4",
    "selfLink": "/api/v1/namespaces/default",
    "uid": "fd497868-e95a-11e7-adce-080027c94384"
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Active"
  }
}

```

Я обнаружил эту конечную точку, когда перешел в `/api`, затем увидел `/api/v1`, который сказал мне, что есть `/api/v1/namespaces`, а уже они указали мне на `/api/v1/namespaces/default`.

Использование Postman для изучения API Kubernetes

Postman (www.getpostman.com) — это хорошо обкатанное приложение для работы с RESTful API. Если вы предпочитаете графический интерфейс, эта утилита покажется вам чрезвычайно полезной.

На рис. 12.1 показаны доступные конечные точки в группе пакета v1 API.

У Postman есть много параметров, и он очень удачно организует информацию. Попробуйте это приложение.

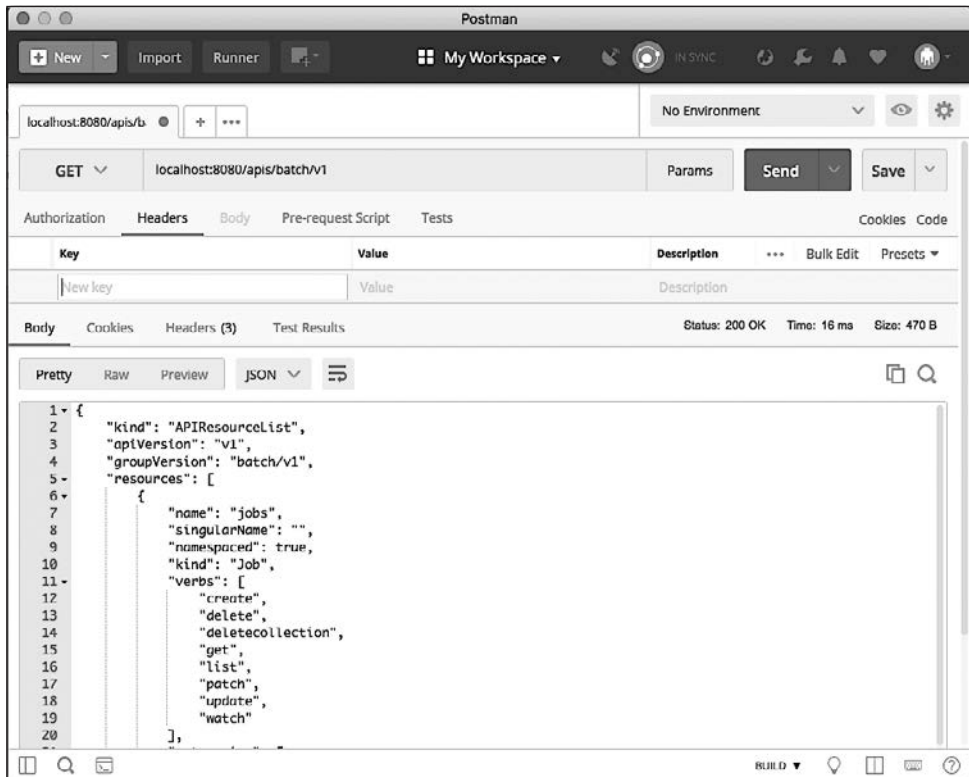


Рис. 12.1

Фильтрация вывода с помощью httpie и jq

Вывод из API иногда оказывается слишком многословным. Часто вам интересно только одно значение из огромного количества ответов JSON. Например, если вы хотите получить имена всех запущенных сервисов, то можете попасть в конечную точку `/api/v1/services`. Однако ответ включает в себя много дополнительной информации, которая нам неинтересна. Вот лишь небольшая часть вывода:

```
$ http http://localhost:8080/api/v1/services
{
  "apiVersion": "v1",
  "items": [
    {
      "metadata": {
```

```

        "creationTimestamp": "2018-03-03T05:18:30Z",
        "labels": {
            "component": "apiserver",
            "provider": "kubernetes"
        },
        "name": "kubernetes",
        ...
    },
    "spec": {
        ...
    },
    "status": {
        "loadBalancer": {}
    }
},
...
],
"kind": "ServiceList",
"metadata": {
    "resourceVersion": "1076",
    "selfLink": "/api/v1/services"
}
}

```

Полный вывод состоит из 121 строки! Посмотрим, как использовать `httpie` и `jq`, чтобы получить полный контроль над выводом и показать только имена сервисов. Для взаимодействия с API REST в командной строке я предпочитаю сервис `httpie.org` на базе `CURL`. `JQ` (stedolan.github.io/jq/), консольный обработчик JSON, отлично подходит для разбора кода в этом формате.

Изучив полный вывод, вы увидите, что имена сервисов находятся в разделах метаданных каждого элемента массива. Выражение `jq`, которое оставляет только `name`, выглядит следующим образом:

```
.items[].metadata.name
```

Вот вывод из командной строки:

```

$ http http://localhost:8080/api/v1/services | jq .items[].metadata.name
"kubernetes"
"kube-dns"
"kubernetes-dashboard"

```

Создание пода с помощью API Kubernetes

API тоже можно использовать для создания, обновления и удаления ресурсов. Допустим, у нас есть следующий манифест пода в `nginx-pod.json`:

```

{
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata": {

```

```

    "name": "nginx",
    "namespace": "default",
    "labels": {
        "name": "nginx"
    }
},
"spec": {
    "containers": [{
        "name": "nginx",
        "image": "nginx",
        "ports": [{"containerPort": 80}]
    }]
}
}

```

Следующая команда создаст под через API:

```
> http POST http://localhost:8080/api/v1/namespaces/default/pods @nginxpod.json
```

Чтобы убедиться в ее успешном выполнении, выберем имя и статус текущих подов. Конечная точка выглядит следующим образом:

```
/api/v1/namespaces/default/pods
```

Выражение `jq` выглядит так:

```
items[].metadata.name,.items[].status.phase
```

Вот полная команда и вывод:

```

> FILTER='.items[].metadata.name,.items[].status.phase'
> http http://localhost:8080/api/v1/namespaces/default/pods | jq $FILTER
"nginx"
"Running"

```

Доступ к API Kubernetes через клиент Python

Изучение API в интерактивном режиме с помощью `httpie` и `jq` — это хорошо, но реальная сила API проявляется, когда вы интегрируете его с другим программным обеспечением.

Проект инкубатора Kubernetes предоставляет полноценную и хорошо документированную клиентскую библиотеку Python `client`. Она доступна по адресу github.com/kubernetes-incubator/client-python.

Сначала убедитесь, что у вас установлен Python (2.7 либо 3.5+). Затем установите пакет Kubernetes:

```
> pip install kubernetes
```

Для общения с кластером Kubernetes к нему нужно сначала подключиться. Начните интерактивную сессию Python:

```
> python
Python 3.6.4 (default, Mar 1 2018, 18:36:42)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Клиент Python может читать вашу конфигурацию Kubectl:

```
>>> from kubernetes import client, config
>>> config.load_kube_config()
>>> v1 = client.CoreV1Api()
```

Или напрямую подключаться к уже запущенному прокси:

```
>>> from kubernetes import client, config
>>> client.Configuration().host = 'http://localhost:8080'
>>> v1 = client.CoreV1Api()
```

Обратите внимание на то, что клиентский модуль обеспечивает методы для доступа к различным версиям групп, таким как CoreV1API.

Анализ группы CoreV1API

Давайте погрузимся в рассмотрение группы CoreV1API. Объект Python содержит 481 публичный атрибут:

```
>>> attributes = [x for x in dir(v1) if not x.startswith('__')]
>>> len(attributes)
481
```

Игнорируйте атрибуты, начинающиеся с двойных подчеркиваний, потому что это специальные методы класса/экземпляра, не связанные с Kubernetes.

Возьмем наугад десять методов и посмотрим, как они выглядят:

```
>>> import random
>>> from pprint import pprint as pp
>>> pp(random.sample(attributes, 10))
['patch_namespaced_pod',
 'connect_options_node_proxy_with_path_with_http_info',
 'proxy_delete_namespaced_pod_with_path',
 'delete_namespace',
 'proxy_post_namespaced_pod_with_path_with_http_info',
 'proxy_post_namespaced_service',
 'list_namespaced_pod_with_http_info',
 'list_persistent_volume_claim_for_all_namespaces',
 'read_namespaced_pod_log_with_http_info',
 'create_node']
```

Очень интересно. Атрибуты начинаются с глагола, например «пролистать» (list), «залатать» (patch) или «прочитать» (read). Во многих из них используется понятие namespace, и у многих есть суффикс with_http_info. Для того чтобы лучше понять, посчитаем все доступные глаголы и посмотрим, в скольких атрибутах

употребляется каждый из них (глагол здесь — первое слово перед подчеркиванием):

```
>>> from collections import Counter
>>> verbs = [x.split('_')[0] for x in attributes]
>>> pp(dict(Counter(verbs)))
{'api': 1,
 'connect': 96,
 'create': 36,
 'delete': 56,
 'get': 2,
 'list': 56,
 'patch': 48,
 'proxy': 84,
 'read': 52,
 'replace': 50}
```

Продолжим работу и посмотрим интерактивную справку для определенного атрибута:

```
>>> help(v1.create_node)
Help on method create_node in module kuber-netes.client.apis.core_v1_api:
```

```
create_node(body, **kwargs) method of kubernetes.client.apis.core_v1_
api.CoreV1Api instance
    create a Node
    This method makes a synchronous HTTP request by default. To make
    an asynchronous
    HTTP request, please pass async=True
    >>> thread = api.create_node(body, async=True)
    >>> result = thread.get()

:param async bool
:param V1Node body: (required)
:param str pretty: If 'true', then the output is pretty printed.
:return: V1Node
         If the method is called asynchronously,
         returns the request thread.
```

Вы можете попробовать и узнать больше об API. Рассмотрим общие операции — перечисление, создание, просмотр и удаление объектов.

Перечисление объектов

Можно перечислить различные типы объектов. Имена методов начинаются с `list_`. Вот пример, в котором перечисляются все пространства имен:

```
>>> for ns in v1.list_namespace().items:
...     print(ns.metadata.name)
...
default
kube-public
kube-system
```


Создание объектов

Чтобы создать объект, необходимо передать параметр `body` методу `create`. Параметр должен быть Python-словарем, эквивалентным файлу конфигурации YAML, который используется с `Kubectrl`. Самый простой способ сделать это — фактически применить YAML, а затем задействовать модуль `YAML Python` (не входит в стандартную библиотеку и устанавливается отдельно), чтобы прочитать файл YAML и загрузить его в словарь. Например, чтобы создать `nginx-deployment` с тремя репликами, можно взять такой файл конфигурации YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Чтобы установить модуль `yaml Python`, введите команду:

```
> pip install yaml
```

Затем следующая программа Python создаст развертывание:

```
from os import path
import yaml
from kubernetes import client, config

def main():
    # Configs can be set in Configuration class directly or using
    # helper utility. If no argument provided, the config will be
    # loaded from default location.
    config.load_kube_config()

    with open(path.join(path.dirname(__file__),
                        'nginx-deployment.yaml')) as f:
        dep = yaml.load(f)
        k8s = client.AppsV1Api()
        status = k8s_beta.create_namespaced_deployment(
            body=dep, namespace="default").status
        print("Deployment created. status='{}'.format(status))

if __name__ == '__main__':
    main()
```

Просмотр объектов

Просмотр объектов — это расширенная возможность. Она реализована с использованием отдельного модуля `watch`. Далее представлен пример того, как посмотреть десять событий пространства имен и вывести их на экран:

```
from kubernetes import client, config, watch

# Configs can be set in Configuration class directly or using helper
utility
config.load_kube_config()
v1 = client.CoreV1Api()
count = 10
w = watch.Watch()
for event in w.stream(v1.list_namespace, _request_timeout=60):
    print(f"Event: {event['type']} {event['object'].metadata.name}")
    count -= 1
    if count == 0:
        w.stop()

print('Done.')
```

Вызов kubectl программно

Если вы не разработчик на языке Python и не хотите напрямую обращаться к REST API, у вас есть другой вариант. Kubectl используется в основном для интерактивной работы в командной строке, но ничто не мешает вам автоматизировать его и вызывать через скрипты и программы. Вот некоторые из преимуществ применения kubectl в качестве вашего клиента API Kubernetes.

- ❑ Легко найти примеры для любого сценария использования.
- ❑ Легко экспериментировать в командной строке, чтобы найти правильную комбинацию команд и аргументов.
- ❑ Kubectl поддерживает вывод в JSON или YAML для проведения быстрого разбора.
- ❑ Аутентификация встроена на уровне конфигурации kubectl.

Запуск Kubectl из Python с помощью модуля subprocess

Я снова работаю с Python, так что можете сравнить официальный клиент Python с написанием своего собственного. У Python есть модуль `subprocess`, способный запускать внешние процессы, такие как kubectl, и захватывать вывод. Вот пример Python 3, который самостоятельно запускает kubectl и отображает начало вывода:

```
>>> import subprocess
>>> out = subprocess.check_output('kubectl').decode('utf-8')
>>> print(out[:276])
```

Kubectl управляет менеджером кластера Kubernetes. Дополнительную информацию можно найти на странице github.com/kubernetes/kubernetes.

Вот несколько основных команд для начинающих:

- ❑ `create` — создать ресурс, используя имя файла или `stdin`;
- ❑ `expose` — взять контроллер репликации, сервис, развертывание или под.

Функция `check_output()` захватывает вывод в виде массива байтов, который для корректного отображения необходимо декодировать в UTF-8. Немного обобщив, мы можем создать удобную функцию `k`, которая принимает параметры, передающиеся в `kubectl`, а затем декодирует вывод и возвращает его:

```
from subprocess import check_output
def k(*args):
    out = check_output(['kubectl'] + list(args))
    return out.decode('utf-8')
Let's use it to list all the running pods in the default namespace:
>>> print(k('get', 'po'))
```

NAME	Ready	Status	Restarts	Age
nginx-deployment-6c54bd5869-9mp2g	1/1	Running	0	18m
nginx-deployment-6c54bd5869-lgs84	1/1	Running	0	18m
nginx-deployment-6c54bd5869-n7468	1/1	Running	0	18m

Это неплохая демонстрация, но все это можно сделать и с помощью `kubectl`. По-настоящему этот подход проявляет себя при использовании параметров структурированного вывода с флагом `-o`. Это позволяет автоматически преобразовать результат в объект языка Python. Вот модифицированная версия функции `k()`, которая в качестве аргумента принимает ключевое слово `use_json` типа `Boolean` (по умолчанию `false`). Если он равен `true`, к команде добавляется параметр `-o json`, благодаря которому вывод формата JSON переводится в объект (словарь) языка Python:

```
from subprocess import check_output
import json

def k(use_json=False, *args):
    cmd = ['kubectl']
    cmd += list(args)
    if use_json:
        cmd += ['-o', 'json']
    out = check_output(cmd)
    if use_json:
        out = json.loads(out)
    else:
        out = out.decode('utf-8')
    return out
```

Этот код возвращает полнофункциональный объект API, по которому можно перемещаться и углубляться так же, как при обращении к REST API напрямую или с использованием официального клиента Python:

```
result = k('get', 'po', use_json=True)
for r in result['items']:
    print(r['metadata']['name'])
nginx-deployment-6c54bd5869-9mp2g
nginx-deployment-6c54bd5869-lgs84
nginx-deployment-6c54bd5869-n7468
```

Попробуем удалить deployment и подождем, пока все поды не исчезнут. Команда удаления kubectl не принимает параметр -o json (хотя она поддерживает -o name), поэтому применим use_json:

```
k('delete', 'deployment', 'nginx-deployment')
while len(k('get', 'po', use_json=True)['items']) > 0:
    print('.')
print('Done.')
Done.
```

Расширение API Kubernetes

Kubernetes — чрезвычайно гибкая платформа. Она позволяет расширять свой API за счет добавления новых типов ресурсов, которые называются пользовательскими. Если этого недостаточно, вы даже можете предоставить свой API-сервер, который интегрируется с API-сервером Kubernetes в рамках механизма под названием *агрегация API*. Что можно сделать с пользовательскими ресурсами? Много. Вы можете с их помощью управлять ресурсами API Kubernetes, которые располагаются за пределами кластера Kubernetes и с которыми связываются ваши поды.

Добавляя эти внешние ресурсы в качестве пользовательских, вы можете получить полноценное представление о своей системе и воспользоваться многими возможностями API Kubernetes, такими как:

- ☐ пользовательские конечные точки CRUD REST;
- ☐ управление версиями;
- ☐ наблюдения;
- ☐ автоматическая интеграция с универсальными инструментами Kubernetes.

Другими вариантами применения пользовательских ресурсов являются метаданные для пользовательских контроллеров и программ автоматизации.

Пользовательские ресурсы, которые впервые появились в Kubernetes 1.7, являются большим шагом вперед по сравнению с устаревшими ресурсами сторонних разработчиков. Давайте посмотрим, что они собой представляют.

Понимание структуры пользовательского ресурса

Чтобы можно было успешно взаимодействовать с API-сервером Kubernetes, сторонние ресурсы должны соответствовать некоторым основным требованиям. Подобно встроенным объектам API, они должны иметь следующие поля:

- ❑ `apiVersion` — `apiextensions.k8s.io/v1beta1`;
- ❑ `metadata` — стандартные метаданные объекта Kubernetes;
- ❑ `kind` — `CustomResourceDefinition`;
- ❑ `spec` — описывает, как ресурс выглядит в API и инструментах;
- ❑ `status` — определяет текущее состояние CRD.

Внутренняя структура спецификации включает в себя следующие поля: группу, имена, область действия, проверку и версию. Статус содержит поля `acceptedNames` и `Conditions`. В следующем разделе я приведу пример, который разъясняет смысл этих полей.

Определение пользовательских ресурсов

Для разработки пользовательского ресурса нужно создать его определение, также известное как CRD. Цель состоит в том, чтобы определение CRD плавно интегрировалось с системой Kubernetes, ее API и инструментами, поэтому нам придется предоставить много информации. Далее приведен пример пользовательского ресурса Candy:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form:
  <plural>.<group>
  name: candies.awesome.corp.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: awesome.corp.com
  # version name to use for REST API: /apis/<group>/<version>
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: candies
    # singular name to be used as an alias on the CLI and for display
    singular: candy
    # kind is normally the CamelCased singular type. Your resource manifests
    # use this.
    kind: Candy
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
      - cn
```

Создадим его:

```
> kubectl create -f crd.yaml
customresourcedefinition "candies.awesome.corp.com" created
```

Обратите внимание на то, что имя метаданных возвращается во множественном числе. Теперь убедимся в том, что оно доступно:

```
> kubectl get crd
NAME AGE
candies.awesome.corp.com 17m
```

Есть также новая конечная точка API для управления новым ресурсом:

```
/apis/awesome.corp.com/v1/namespaces/<namespace>/candies/
```

Для доступа к ней воспользуемся нашим кодом на языке Python:

```
>>> config.load_kube_config()
>>> print(k('get', 'thirdpartyresources'))
NAME AGE
candies.awesome.corp.com 24m
```

Интеграция пользовательских ресурсов

После создания объекта `CustomResourceDefinition` вы можете создавать собственные ресурсы этого типа, в частности `Candy` (это `Candy` в формате `CamelCase`). `Candy` может содержать нужные поля с произвольным JSON-кодом. В следующем примере настраивается поле `flavor` в объекте `Candy`. Поле `apiVersion` основано на группе и полях версий спецификации CRD:

```
apiVersion: "awesome.corp.com/v1"
kind: Candy
metadata:
  name: chocolatem
spec:
  flavor: "sweeeeeeeet"
```

Вы можете добавить произвольные поля в свои ресурсы. Это могут быть любые значения формата JSON. Обратите внимание на то, что эти поля не определены в CRD. Разные объекты могут иметь разные поля. Приступим к созданию:

```
> kubectl create -f candy.yaml
candy "chocolate" created
```

На этом этапе утилита `kubectl` способна работать с объектами `Candy` точно так же, как с обычными встроенными объектами. Обратите внимание, что при использовании `kubectl` имена ресурсов не зависят от регистра:

```
$ kubectl get candies
NAME AGE
chocolate 2m
```

Мы также можем просмотреть необработанные данные JSON, применив стандартный флаг `-o json`. На этот раз я возьму короткое имя `cn`:

```
> kubectl get cn -o json
{
```

```

"apiVersion": "v1",
"items": [
  {
    "apiVersion": "awesome.corp.com/v1",
    "kind": "Candy",
    "metadata": {
      "clusterName": "",
      "creationTimestamp": "2018-03-07T18:18:42Z",
      "name": "chocolate",
      "namespace": "default",
      "resourceVersion": "4791773",
      "selfLink": "/apis/awesome.corp.com/v1/namespaces/default/
        candies/chocolate",
      "uid": "f7a6fd80-2233-11e8-b432-080027c94384"
    },
    "spec": {
      "flavor": "sweeeeeeeet"
    }
  },
],
"kind": "List",
"metadata": {
  "resourceVersion": "",
  "selfLink": ""
}
}

```

Финализация пользовательских ресурсов

Пользовательские ресурсы поддерживают финализаторы, как и стандартные объекты API. Финализатор — это механизм, в котором объекты не удаляются немедленно, им приходится ждать специальных контроллеров, работающих в фоновом режиме и наблюдающих за запросами на удаление. Контроллер способен применять любые параметры очистки, а затем удалять свой финализатор из целевого объекта. К одному объекту можно применить несколько финализаторов. Kubernetes будет ждать, пока все они будут удалены, и только затем удалит объект. Финализаторы в метаданных — это просто произвольные строки, которые определяет соответствующий контроллер. Вот пример с объектом `Candy`, имеющим два финализатора, `eat-me` и `drink-me`:

```

apiVersion: "awesome.corp.com/v1"
kind: Candy
metadata:
  name: chocolate
  finalizers:
    - eat-me
    - drink-me
spec:
  flavor: "sweeeeeeeet"

```

Утверждение пользовательских ресурсов

В CRD можно добавить любое поле. Из-за этого могут возникнуть некорректные определения. В Kubernetes 1.9 появился механизм проверки CRD на основе схемы OpenAPI V3. Он все еще находится в стадии бета-тестирования и отключается с помощью параметра `gate` при запуске сервера API:

```
--feature-gates=CustomResourceValidation=false
```

В CRD вы добавляете раздел проверки в спецификацию:

```
validation:
  openAPIV3Schema:
    properties:
      спец:
        properties:
          cronSpec:
            type: string
            pattern: '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+)?){4}$'
          replicas:
            type: integer
            minimum: 1
            maximum: 10
```

Если вы попытаетесь создать объекты, которые противоречат спецификации, то получите сообщение об ошибке. Подробнее о схеме OpenAPI можете прочитать, зайдя на страницу bit.ly/2FsBfWA.

Агрегация API-серверов

Спецификации CRD прекрасно подходят, когда вам нужно выполнить лишь некие CRUD-операции со своими собственными типами. Вы можете просто опереться на API-сервер Kubernetes, который будет хранить ваши объекты и предоставлять поддержку API и интеграцию с инструментами, такими как `kubectl`. Можете запускать контроллеры, которые следят за вашими объектами, и выполнять некоторые операции, когда те создаются, обновляются или удаляются. Но CRD имеют ограничения. Если вам нужны более сложные функции и настройки, используйте агрегацию API-серверов и напишите собственный API-сервер, которому будут делегироваться обязанности API-сервера Kubernetes.

Ваш API-сервер будет задействовать тот же механизм, что и API-сервер Kubernetes. Вот некоторые из расширенных возможностей.

- ☐ Управление хранением ваших объектов.
- ☐ Мультиверсионность.
- ☐ Пользовательские операции помимо CRUD (например, `exes` или `scale`).
- ☐ Использование данных в формате `protobuf`.

Написание расширяющего API-сервера — это нетривиальная задача. Если вы решите, что вам нужна вся эта мощь, я рекомендую воспользоваться проектом API Builder: github.com/kubernetes-incubator/apiserver-builder.

API Builder предоставляет следующие возможности.

- ❑ Сборка определений всех типов, контроллеров, тестов, а также документации.
- ❑ Введение расширенного управляющего уровня, который можно запускать локально, внутри Minikube или в реальном удаленном кластере.
- ❑ Создание сгенерированных контроллеров, которые смогут отслеживать и обновлять объекты API.
- ❑ Добавление ресурсов (включая подресурсы).
- ❑ Наличие значений по умолчанию, которые при необходимости можно переопределить.

Использование каталога сервисов

Проект каталога сервисов Kubernetes позволяет без труда и безболезненно интегрировать любые внешние сервисы, поддерживающие спецификацию Open Service Broker API: github.com/openservicebrokerapi/servicebroker.

Цель Open Service Broker API заключается в том, чтобы делать доступными внешние сервисы для любой облачной среды через стандартную спецификацию с поддержкой документации и комплексного набора тестов. Это позволяет провайдерам реализовывать единую спецификацию и поддерживать несколько облачных сред. На сегодняшний день поддерживаются такие среды, как Kubernetes и CloudFoundry. Идет работа над широким промышленным внедрением проекта.

Каталог сервисов особенно полезен для интеграции сервисов, предоставляемых провайдерами облачных платформ. Вот примеры таких сервисов:

- ❑ Microsoft Azure Cloud Queue;
- ❑ Amazon Simple Queue Service;
- ❑ Google Cloud Pub/Sub.

Такая способность — благо для организаций, приверженных облаку. Вы получаете возможность создавать свою систему на Kubernetes, но вам не нужно развертывать каждый сервис, управлять им и обслуживать его в своем кластере самостоятельно. Отдав все это на откуп провайдеру облака, вы будете наслаждаться глубиной интеграцией и сосредоточиться на своем приложении.

Каталог услуг может сделать ваш кластер Kubernetes полностью автономным, позволяя предоставлять ресурсы облака через сервис-брокеров. Еще не все готово, но направление многообещающее.

На этом завершаем обсуждение доступа к Kubernetes и его расширения извне. Далее рассмотрим настройку внутренней работы Kubernetes через дополнения.

Написание дополнений Kubernetes

Здесь мы погрузимся в платформу Kubernetes и научимся использовать ее знаменитую гибкость и расширяемость. Узнаем о различных аспектах, которые можно настроить с помощью дополнений, и о том, как реализовать такие дополнения и интегрировать их с Kubernetes.

Создание пользовательского дополнения-планировщика

Kubernetes позиционируется как система планирования контейнеров и управления ими. Таким образом, планировщик — это наиболее важный компонент Kubernetes. Kubernetes поставляется с планировщиком по умолчанию, но позволяет писать дополнительные планировщики. Чтобы создать собственный планировщик, нужно понять, что он делает, как упакован, как развернуть планировщик и интегрировать его. Исходный код планировщика доступен здесь: github.com/kubernetes/kubernetes/tree/master/pkg/scheduler.

Далее мы подробно рассмотрим исходный код, а также типы данных и алгоритмы.

Архитектура планировщика Kubernetes

Задача планировщика — найти узел для вновь созданных или перезапущенных подов, создать привязку на API-сервере и запустить его там. Если планировщик не может найти подходящий узел для пода, он останется в состоянии ожидания.

Планировщик

Большая часть работы планировщика довольно общая — определить, какие поды нуждаются в планировании, обновить их состояние и запустить на выбранном узле. Настраиваемым аспектом является сопоставление подов с узлами. Команда Kubernetes признала необходимость пользовательского планирования, а общий планировщик может быть сконфигурирован с применением разных алгоритмов планирования.

Основной тип данных — `struct Scheduler`, он содержит `Config struct` с множеством свойств (скоро это будет заменено интерфейсом `configurator`):

```
type Scheduler struct {
    config *Config
}
```

Вот `Config struct`:

```
type Config struct {
    SchedulerCache schedulercache.Cache
    Ecache *core.EquivalenceCache
}
```

```

NodeLister algorithm.NodeLister
Algorithm algorithm.ScheduleAlgorithm
GetBinder func(pod *v1.Pod) Binder
PodConditionUpdater PodConditionUpdater
PodPreemptor PodPreemptor
NextPod func() *v1.Pod
WaitForCacheSync func() bool
Error func(*v1.Pod, error)
Recorder record.EventRecorder
StopEverything chan struct{}
VolumeBinder *volumebinder.VolumeBinder
}

```

Большинство этих свойств — интерфейсы, поэтому вы можете настроить планировщик для обеспечения нестандартной функциональности. Особую роль в настройке планирования подов играет алгоритм планировщика.

Регистрация провайдера алгоритма

Планировщик имеет концепцию провайдера алгоритмов и алгоритма. Вместе они позволяют задействовать существенные функциональные возможности встроенного планировщика, чтобы заменить основной алгоритм планирования.

Провайдер алгоритмов позволяет регистрировать новые провайдеры алгоритмов с помощью фабрики. В Kubernetes уже зарегистрирован один пользовательский провайдер под названием `ClusterAutoScalerProvider`. Позднее мы увидим, как планировщик определяет, какой провайдер алгоритмов использовать. Ключевой файл выглядит так: github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/algorithmprovider/defaults/defaults.go.

Функция `init()` делает вызов `registerAlgorithmProvider()`, который вы должны расширить, чтобы добавить провайдер алгоритма к провайдерам по умолчанию и `autoscaler`:

```

func registerAlgorithmProvider(predSet, priSet sets.String) {
    // Регистрирует провайдеры алгоритмов. По умолчанию мы используем
    // 'DefaultProvider', но пользователь может указать другой провайдер
    // с помощью флага.
    factory.RegisterAlgorithmProvider(factory.DefaultProvider, predSet,
        priSet)
    // Алгоритм, подходящий для автомасштабирования кластера.
    factory.RegisterAlgorithmProvider(ClusterAutoscalerProvider, predSet,
        copyAndReplace(priSet, "LeastRequestedPriority", "MostRequestedPriority"))
}

```

Помимо регистрации провайдера, необходимо зарегистрировать подходящий предикат и функцию приоритета, с применением которых фактически происходит планирование.

Вы можете использовать фабричные функции `RegisterFitPredicate()` и `RegisterPriorityFunction2()`.

Конфигурирование планировщика

Алгоритм планировщика — это часть конфигурации. Пользовательские планировщики могут реализовать интерфейс `ScheduleAlgorithm`:

```
type ScheduleAlgorithm interface {
    Schedule(*v1.Pod, NodeLister) (selectedMachine string, err error)
    Preempt(*v1.Pod, NodeLister, error) (selectedNode *v1.Node,
                                         preemptedPods []*v1.Pod,
                                         cleanupNominatedPods []*v1.Pod,
                                         err error)

    Predicates() map[string]FitPredicate
    Prioritizers() []PriorityConfig
}
```

При запуске планировщика в качестве аргумента командной строки можно указать имя пользовательского планировщика или провайдера пользовательского алгоритма. Если ни один из них не предоставлен, будет использоваться провайдер алгоритма по умолчанию. Аргументами командной строки для планировщика являются `--algorithm-provider` и `--scheduler-name`.

Упаковка планировщика

Пользовательский планировщик работает как под внутри того же кластера Kubernetes, за который он отвечает. Он должен быть упакован как образ контейнера. Давайте возьмем копию стандартного планировщика Kubernetes для демонстрационных целей. Мы можем собрать Kubernetes из исходного кода, чтобы получить образ планировщика:

```
git clone https://github.com/kubernetes/kubernetes.git
cd kubernetes
make
```

Создайте следующий Dockerfile:

```
FROM busybox
ADD ./_output/bin/kube-scheduler /usr/local/bin/kube-scheduler
```

Используйте его для создания типа образа Docker:

```
docker build -t custom-kube-scheduler:1.0 .
```

Наконец, поместите образ в реестр контейнеров. Здесь я буду использовать DockerHub. Необходимо создать учетную запись на DockerHub и войти в систему, прежде чем поместить образ:

```
> docker login
> docker push g1g1/custom-kube-scheduler
```

Обратите внимание на то, что я создал планировщик локально, а в Dockerfile просто скопировал его с хоста в образ. Это работает при развертывании на той же ОС, с которой вы работаете. Если это не так, лучше вставить команды сборки в Dockerfile. Недостатком этого подхода является то, что вам придется поместить в образ весь код Kubernetes.

Развертывание пользовательского планировщика

Теперь, когда образ планировщика собран и доступен в реестре, нужно создать для него развертывание Kubernetes. Планировщик, конечно, критически важен, поэтому мы можем использовать саму платформу Kubernetes, чтобы гарантировать, что он всегда будет работать. Следующий файл YAML определяет развертывание с одной репликой и несколькими дополнительными возможностями, такими как проверка жизнеспособности и готовности:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: custom-scheduler
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      containers:
      - command:
        - /usr/local/bin/kube-scheduler
        - --address=0.0.0.0
        - --leader-elect=false
        - --scheduler-name=custom-scheduler
        image: glg1/custom-kube-scheduler:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
          initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10251
        resources:
          requests:
            cpu: '0.1'
```

Имя планировщика (здесь `custom-scheduler`) имеет значение и должно быть уникальным. Его будут использовать позже для связывания подов с планировщиком, чтобы запланировать их. Обратите внимание на то, что пользовательский планировщик принадлежит пространству имен `kube-system`.

Запуск еще одного пользовательского планировщика в кластере

Запустить еще один пользовательский планировщик так же просто, как и создать развертывание. В этом красота инкапсулированного подхода. Ниже Kubernetes запустит второй планировщик (что неординарно само по себе), не зная при этом, что на самом деле происходит. Kubernetes просто развернет его как любой другой под, только в данном случае экземпляр пода является пользовательским планировщиком:

```
$ kubectl create -f custom-scheduler.yaml
```

Проверим, работает ли под планировщика:

```
$ kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
....				
custom-scheduler-7cfc49d749-lwzxj	1/1	Running	0	2m
...				

Пользовательский планировщик запущен.

Назначение подов пользовательскому планировщику

Итак, пользовательский планировщик работает вместе с планировщиком по умолчанию. Но как Kubernetes выбирает, какой из них использовать, когда под нуждается в планировании? Ответ такой: решает под, а не Kubernetes. Спецификация пода имеет необязательное поле имени планировщика. Если его нет, берется планировщик по умолчанию, в противном случае — указанный планировщик. Именно поэтому имена пользовательских планировщиков должны быть уникальными. Имя планировщика по умолчанию — `default-scheduler` на случай, если вы хотите указать его явно. Вот определение пода, который будет планироваться с применением планировщика по умолчанию:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
  labels:
    name: some-pod
spec:
  containers:
  - name: some-container
    image: gcr.io/google_containers/pause:2.0
```

Чтобы `custom-scheduler` планировал этот под, измените его спецификацию следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
```

```
labels:
  name: some-pod
spec:
  schedulerName: custom-scheduler
  containers:
  - name: some-container
    image: gcr.io/google_containers/pause:2.0
```

Проверим, запланированы ли наши поды с помощью пользовательского планировщика

Существует два основных способа проверки того, что поды запланированы правильным планировщиком. Способ первый: создавайте поды, которые должны быть запланированы с помощью пользовательского планировщика, перед развертыванием последнего. Контейнеры останутся в состоянии ожидания. Затем разверните пользовательский планировщик, и отложенные поды будут запланированы и запущены.

Способ второй: проверьте журналы событий и просмотрите запланированные события с помощью команды:

```
$ kubectl get events
```

Использование веб-хуков для контроля доступа

Платформа Kubernetes всегда предоставляла возможность настроить контроль доступа. В этой системе контроль доступа составляют аутентификация, авторизация и управление входом. В ранних версиях это было сделано с помощью дополнений, которые требовали программирования на языке Go, установки в ваш кластер, регистрации и других процедур. Теперь Kubernetes позволяет настраивать веб-хуки для проверки подлинности, авторизации и контроля доступа.

Веб-хуки для аутентификации

Kubernetes позволяет расширять процесс аутентификации путем внедрения веб-хуков для Bearer Token. Для этого требуется информация о том, как получить доступ к сервису удаленной аутентификации, и о длительности принятия решения об аутентификации (по умолчанию — 2 минуты).

Чтобы предоставить данную информацию и включить аутентификационные веб-хуки, запустите API-сервер со следующими аргументами командной строки:

- ❑ `--runtime-config=authentication.k8s.io/v1beta1=true;`
- ❑ `--authentication-token-webhook-config-file;`
- ❑ `--authentication-token-webhook-cache-ttl.`

Файл конфигурации использует формат kubeconfig. Вот пример:

```
clusters:
- name: remote-authentication-service
  cluster:
    certificate-authority: /path/to/ca.pem
    server: https://example.com/authenticate

users:
- name: k8s-api-server
  user:
    client-certificate: /path/to/cert.pem
    client-key: /path/to/key.pem

current-context: webhook
contexts:
- context:
    cluster: remote-authentication-service
    user: k8s-api-sever
  name: webhook
```

Обратите внимание на то, что клиентский сертификат и ключ должны быть предоставлены Kubernetes для взаимной аутентификации в сервисе удаленной аутентификации.

Кэш TTL полезен, потому что часто пользователи делают несколько последовательных запросов в Kubernetes. Наличие кэшированного решения для проверки подлинности способно сэкономить много циклов в сервисе удаленной аутентификации.

Когда HTTP-запрос приходит, Kubernetes извлекает токен-носитель из заголовков и отправляет JSON-запрос `TokenReview` в сервис удаленной аутентификации через веб-хуки:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "<bearer token from original request headers>"
  }
}
```

Сервис удаленной аутентификации сообщит о принятом решении. Состояние идентификации будет либо истинным, либо ложным. Вот пример успешной проверки подлинности:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      "username": "gigi@gg.com",
```



```

    "uid": "42",
    "groups": [
      "developers",
    ],
    "extra": {
      "extrafield1": [
        "extravalue1",
        "extravalue2"
      ]
    }
  }
}

```

Отклоненный ответ гораздо более краток:

```

{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false
  }
}

```

Веб-хуки для авторизации

Веб-хуки для авторизации очень похожи на аутентификационные веб-хуки. Для них требуется только файл конфигурации в том же формате, что и файл конфигурации веб-хуков аутентификации. Авторизация не кэшируется, поскольку, в отличие от проверки подлинности, один и тот же пользователь может делать много запросов к различным конечным точкам API с разными параметрами, а решения по авторизации могут быть разными, поэтому кэширование — неправильный вариант.

Для настройки веб-хуков нужно передать API-серверу следующие аргументы командной строки:

- ❑ `--runtime-config=authorization.k8s.io/v1beta1=true;`
- ❑ `--authorization-webhook-config-file=<имя_файла_конфигурации>.`

Когда запрос пройдет аутентификацию, Kubernetes отправит сервису удаленной авторизации объект JSON `SubjectAccessReview`. Он будет содержать запрос пользователя, а также запрошенные ресурсы и другие атрибуты запроса:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "awesome-namespace",
      "verb": "get",

```

```

    "group": "awesome.example.org",
    "resource": "pods"
  },
  "user": "gigi@gg.com",
  "group": [
    "group1",
    "group2"
  ]
}

```

Запрос будет разрешен:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}

```

Или запрещен (с указанием причины):

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the namespace"
  }
}

```

Пользователь может иметь доступ к ресурсу, но не к другим атрибутам, таким как `/api`, `/apis`, `/metrics`, `/resetMetrics`, `/logs`, `/debug`, `/healthz`, `/swagger-ui/`, `/swaggerapi/`, `/ui` и `/version`.

Вот как запросить доступ к журналам:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "nonResourceAttributes": {
      "path": "/logs",
      "verb": "get"
    },
    "user": "gigi@gg.com",
    "group": [
      "group1",
      "group2"
    ]
  }
}

```

Веб-хуки для контроля входа

Динамический контроль входа также поддерживает веб-хуки. Чтобы включить универсальный веб-хук для контроля доступа, нужно передать API-серверу следующие параметры командной строки:

- ❑ `--admission-control=GenericAdmissionWebhook;`
- ❑ `--runtime-config=admissionregistration.k8s.io/v1alpha1.`

Конфигурирование контроллера веб-хуков входа на лету. Веб-хуки аутентификации и авторизации должны быть настроены при запуске API-сервера. Веб-хуки контроля входа могут быть настроены динамически, с созданием объектов `Externaladmissionhookconfiguration`:

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: ExternalAdmissionHookConfiguration
metadata:
  name: example-config
externalAdmissionHooks:
- name: pod-image.k8s.io
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
  failurePolicy: Ignore
  clientConfig:
    caBundle: <pem encoded ca cert that signs the server cert
              used by the webhook>
  service:
    name: <имя_клиентского_сервиса>
    namespace: <пространство_имен_клиентского_сервиса>
```

Предоставление пользовательских показателей для горизонтального автомасштабирования подов

До Kubernetes 1.6 пользовательские показатели были реализованы как модель Heapster. В Kubernetes 1.6 постепенно начал созревать новый API для пользовательских показателей. Начиная с Kubernetes 1.9 он включен по умолчанию. Пользовательские показатели зависят от агрегации API. Рекомендуемый путь — начать с шаблона API-сервера с поддержкой пользовательских показателей — его можно взять отсюда: github.com/kubernetes-incubator/custom-metrics-apiserver.

Затем вы реализуете интерфейс `CustomMetricsProvider`:

```
type CustomMetricsProvider interface {
    GetRootScopedMetricByName(groupResource schema.GroupResource,
                             name string,
                             metricName string)
    (*custom_metrics.MetricValue, error)
    GetRootScopedMetricBySelector(groupResource schema.GroupResource,
                                  selector labels.Selector,
                                  metricName string)
    (*custom_metrics.MetricValueList, error)
    GetNamespacedMetricByName(groupResource schema.GroupResource,
                              namespace string,
                              name string,
                              metricName string)
    (*custom_metrics.MetricValue, error)
    GetNamespacedMetricBySelector(groupResource schema.GroupResource,
                                  Namespace string,
                                  Selector labels.Selector,
                                  metricName string)
    (*MetricValueList, error)

    ListAllMetrics() []MetricInfo
}
```

Добавление в Kubernetes пользовательского хранилища

Дополнения томов — еще один тип дополнений. До Kubernetes 1.8 вам пришлось бы написать дополнение `Kublet`, которое требует реализации, регистрации в Kubernetes и связывания с `kubelet`. В Kubernetes 1.8 появилось намного более универсальное дополнение, `FlexVolume`. Версия Kubernetes 1.9 сделала еще один шаг вперед благодаря новому *интерфейсу контейнерных хранилищ (CSI)*.

Преимущества `FlexVolume`

Дополнения томов Kubernetes предназначены для поддержки определенного типа хранилища или провайдера хранилища. Существует множество дополнений томов, мы рассмотрели их в главе 7. Существующих дополнений томов более чем достаточно для большинства пользователей, но если вам нужно интегрироваться с системой хранения данных, которая не поддерживается, вы должны реализовать собственное дополнение томов, что довольно сложно. Если хотите, чтобы его приняли в качестве официального дополнения Kubernetes, придется пройти скрупулезный процесс утверждения.

Дополнение `FlexVolume` предоставляет другой путь. Это дополнение общего характера, которое позволяет подключить неподдерживаемый сервер для хранения без глубокой интеграции с самой платформой Kubernetes.

FlexVolume позволяет добавлять произвольные атрибуты в спецификацию и взаимодействует с вашим сервером через интерфейс вызовов, который включает в себя следующие операции:

- ❑ **Attach** — присоединяет том к узлу Kubernetes kubelet;
- ❑ **Detach** — отсоединяет том от узла Kubernetes kubelet;
- ❑ **Mount** — монтирует присоединенный том;
- ❑ **Unmount** — отключает присоединенный том.

Каждая операция реализуется серверным драйвером в виде исполняемого файла, который FlexVolume запускает в нужное время. Драйвер должен быть установлен в `/usr/libexec/kubernetes/kubeletplugins/volume/exec/<поставщик>~<драйвер>/<драйвер>`.

Преимущества CSI

FlexVolume позволяет подключать сторонние дополнения, но для этого по-прежнему требуются собственно дополнение FlexVolume и несколько громоздкая модель установки и вызова. CSI значительно улучшит ситуацию, заставив поставщика реализовать дополнение самостоятельно. Хорошая новость состоит в том, что вам как разработчику не нужно создавать и поддерживать эти дополнения. Поставщик решений для хранения данных несет ответственность за внедрение и обслуживание CSI, и в его интересах сделать все максимально надежно, чтобы люди не выбирали другое решение.

Резюме

В этой главе мы рассмотрели три основные темы: работу с API Kubernetes, расширение API Kubernetes и создание дополнений Kubernetes. API Kubernetes поддерживает спецификацию OpenAPI и служит отличным примером дизайна REST API, который соответствует всем современным практикам. Этот API очень последовательный, хорошо организованный и документированный, но он большой, и его нелегко понять. Вы можете получить доступ к API напрямую через REST по HTTP, используя клиентские библиотеки, включая официальный клиент на языке Python, и даже вызывая `kubectl`.

Расширение Kubernetes API содержит определение ваших собственных пользовательских ресурсов и, возможно, расширение самого API-сервера посредством агрегации API. Пользовательские ресурсы наиболее эффективны при объединении их с пользовательскими дополнениями или контроллерами, когда вы запрашиваете и обновляете их извне.

Дополнения и веб-хуки являются основой дизайна Kubernetes. Kubernetes всегда предназначалась для того, чтобы удовлетворять любые потребности пользователей. Мы рассмотрели различные дополнения и веб-хуки, которые вы

можете написать, и поговорили о том, как регистрировать и легко интегрировать их с Kubernetes.

Мы также рассмотрели пользовательские показатели и даже способы расширения Kubernetes с помощью сторонних хранилищ.

На этом этапе вы должны хорошо знать основные механизмы расширения, настройки и управления Kubernetes через доступ к API, пользовательские ресурсы и дополнения. У вас есть отличная возможность воспользоваться всем этим, чтобы расширить существующие функции Kubernetes и адаптировать их к своим потребностям и системам.

В следующей главе мы рассмотрим Helm — менеджер пакетов Kubernetes и его схемы. Как вы, вероятно, поняли, развертывание и настройка сложных систем на Kubernetes далеко не просты. Helm позволяет группировать связку манифестов в схему, которую можно установить как единое целое.

13

Работа с диспетчером пакетов Kubernetes

В этой главе мы рассмотрим Helm — диспетчер пакетов Kubernetes. Каждая успешная и важная платформа должна иметь хорошую систему пакетирования. Helm был разработан компанией Deis (приобретена Microsoft в апреле 2017 года) и позже вносил свой вклад непосредственно в проект Kubernetes. Для начала мы поговорим о мотивах создания Helm, его архитектуре и компонентах. Затем вы получите практический опыт и узнаете, как применять Helm и его схемы в Kubernetes. Сюда относятся поиск, установка, настройка, удаление схем и управление ими. И последнее, но не менее важное: мы обсудим создание собственных схем и управление версиями, зависимостями и шаблонами.

Будут рассмотрены следующие темы.

- ❑ Знакомство с Helm.
- ❑ Использование Helm.
- ❑ Создание собственных схем.

Знакомство с Helm

Платформа Kubernetes предоставляет множество способов организации и оркестрации ваших контейнеров во время выполнения, но ей не хватает организации на более высоком уровне группировки наборов образов. Вот здесь нам и пригодится Helm. В этом разделе мы поговорим о том, зачем нужен Helm, обсудим его архитектуру и компоненты и расскажем об особенностях перехода с Helm Classic на Helm.

Преимущества

Helm обеспечивает:

- ❑ управление сложностью;
- ❑ легкость обновления;
- ❑ простой обмен пакетами;
- ❑ безопасные откаты.

Схемы могут описывать даже самые сложные приложения, обеспечивать их повторную установку и служить единой точкой полномочий. Локальные обновления и пользовательские хуки позволяют легко обновлять файлы. Вы можете легко обмениваться схемами с другими пользователями, управлять их версиями и размещать их на публичных или частных серверах. Когда вам нужно откатить последние обновления, Helm предоставляет единственную команду для отката целого набора изменений инфраструктуры.

Архитектура

Helm предназначен для выполнения следующих действий:

- ❑ создания новых схем с нуля;
- ❑ упаковки схем в архивные файлы (tgz);
- ❑ взаимодействия с репозиториями схем, где последние хранятся;
- ❑ установки схем в существующий кластер Kubernetes и их удаления оттуда;
- ❑ управления циклом выпуска схем, которые были установлены с помощью Helm.

Для достижения этих целей Helm использует клиент-серверную архитектуру.

Компоненты

Helm имеет серверный компонент, который выполняется на вашем кластере Kubernetes, и клиентский компонент, запускаемый на локальном компьютере.

Сервер Tiller

Сервер отвечает за управление релизами. Он взаимодействует с клиентами Helm, а также API-сервером Kubernetes. Его основные функции:

- ❑ прослушивание входящих запросов от клиента Helm;
- ❑ объединение схемы и конфигурации для сборки релиза;
- ❑ установка схем в Kubernetes;
- ❑ отслеживание очередного релиза;
- ❑ обновление и удаление схем путем взаимодействия с Kubernetes.

Клиент Helm

Вы устанавливаете клиент Helm на свой компьютер. Он отвечает:

- ❑ за разработку локальных схем;
- ❑ управление репозиториями;

- ❑ взаимодействие с сервером Tiller;
- ❑ отправку схем для установки;
- ❑ запрос информации о релизах;
- ❑ запрос обновлений или удаление существующих релизов.

Использование Helm

Helm — это многофункциональная система управления пакетами, которая позволяет выполнять все необходимые шаги по управлению приложениями, установленными в вашем кластере. Давайте закатаем рукава и продолжим.

Установка Helm

Установка Helm предусматривает установку клиента и сервера. Helm реализован на языке Go, поэтому один и тот же исполняемый файл может служить как клиентом, так и сервером.

Установка клиента Helm

У вас должен быть правильно сконфигурирован `kubectl`, чтобы можно было связаться с кластером Kubernetes, потому что клиент Helm использует конфигурацию `kubectl` для общения с сервером Helm (Tiller).

Helm предоставляет собранные версии для всех платформ: github.com/kubernetes/helm/releases/latest.

В Windows также можно использовать менеджер пакетов `chocolatey`, но он может немного отставать от официальной версии:

`chocolatey.org/packages/kubernetes-helm/<версия>`.

В macOS и Linux клиент устанавливается из скрипта:

```
$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get
> get_helm.sh
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

В macOS X можете использовать также Homebrew:

```
brew install kubernetes-helm
```

Установка сервера Tiller

Tiller обычно работает внутри вашего кластера. Для разработки иногда легче запускать Tiller локально.

Установка Tiller в кластере

Самый простой способ установить Tiller — с компьютера, на котором установлен клиент Helm. Выполните следующую команду:

```
helm init
```

Это приведет к инициализации как клиента, так и сервера Tiller на удаленном кластере Kubernetes. Когда установка будет завершена, у вас будет запущен модуль Tiller в пространстве имен `kube-system` вашего кластера:

```
> kubectl get po --namespace=kube-system -l name=tiller
NAME                                READY  STATUS   RESTARTS   AGE
tiller-deploy-3210613906-2j5sh  1/1    Running  0           1m
```

Вы также можете запустить `helm version`, чтобы проверить версию клиента и сервера:

```
> helm version
Client: &version.Version{SemVer:"v2.2.3",
GitCommit:"1402a4d6ec9fb349e17b912e32fe259ca21181e3", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.2.3",
GitCommit:"1402a4d6ec9fb349e17b912e32fe259ca21181e3", GitTreeState:"clean"}
```

Установка Tiller локально

Если вы хотите запустить Tiller локально, его следует сначала собрать. Это можно сделать в Linux и macOS:

```
> cd $GOPATH
> mkdir -p src/k8s.io
> cd src/k8s.io
> git clone https://github.com/kubernetes/helm.git
> cd helm
> make bootstrap build
```

Цель `bootstrap` попытается установить зависимости, перестроить дерево `vendor/` и проверить конфигурацию.

Цель `build` скомпилирует Helm и поместит его в каталог `bin/helm`. Tiller также компилируется и устанавливается в каталог `bin/tiller`.

Теперь можете просто запустить `bin/tiller`. Tiller подключится к кластеру Kubernetes через вашу конфигурацию `kubectl`.

Вы должны сообщить клиенту Helm, что следует подключиться к локальному серверу Tiller. Сделайте это, установив переменную окружения:

```
> export HELM_HOST=localhost:44134
```

В противном случае передайте подходящий адрес в качестве аргумента командной строки: `--host localhost:44134`.

Использование альтернативного серверного хранилища

В Helm 2.7.0 появилась возможность хранить информацию о выпуске в виде *секретов*. Более ранние версии всегда сохраняли информацию о выпуске в ConfigMaps. Секреты повышают безопасность схем. Это дополнение к общему шифрованию статических данных в Kubernetes. Чтобы использовать серверные секреты, нужно запустить Helm с помощью следующей команды:

```
> helm init --override
'spec.template.spec.containers[0].command'='{/tiller,--storage=secret}'
```

Поиск схем

Чтобы установить полезные приложения и программное обеспечение с помощью Helm, нужно сначала найти их схемы. В этом случае используется команда `helm search`. Helm по умолчанию ищет официальный репозиторий схем Kubernetes `chart repository`, который называется `stable`:

```
> helm search
```

NAME	VERSION	DESCRIPTION
stable/acs-engine-autoscaler	2.1.1	Scales worker nodes within agent pools
stable/aerospike	0.1.5	A Helm chart for Aerospike in Kubernetes
stable/artifactory	6.2.4	Universal Repository Manager supporting all maj...
stable/aws-cluster-autoscaler	0.3.2	Scales worker nodes within autoscaling groups.
stable/buildkite	0.2.0	Agent for Buildkite
stable/centrifugo	2.0.0	Centrifugo is a real-time messaging server.
stable/chaoskube	0.6.1	Chaoskube periodically kills random pods in you...
stable/chronograf	0.4.0	Open-source web application written in Go and R..
stable/cluster-autoscaler	0.3.1	Scales worker nodes within autoscaling groups.

В официальном репозитории имеется богатая библиотека схем, которая подобрала в себя все современные открытые системы мониторинга, специальные вспомогательные сервисы Kubernetes и множество других предложений, таких как сервер Minecraft. Можете искать конкретные схемы. Например, давайте поищем схемы, содержащие `kube` в имени или описании:

```
> helm search kube
```

NAME	VERSION	DESCRIPTION
stable/chaoskube	0.6.1	Chaoskube periodically kills random pods in you...

stable/kube-lego	0.3.0	Automatically requests certificates from Let's ...
stable/kube-ops-view	0.4.1	Kubernetes Operational View - read-only system ...
stable/kube-state-metrics	0.5.1	Install kube-state-metrics to generate and expo...
stable/kube2iam	0.6.1	Provide IAM credentials to pods based on annota...
stable/kubed	0.1.0	Kubed by AppsCode – Kubernetes daemon
stable/kubernetes-dashboard	0.4.3	General-purpose web UI for Kubernetes clusters
stable/sumokube	0.1.1	Sumologic Log Collector
stable/aerospike	0.1.5	A Helm chart for Aerospike in Kubernetes
stable/coredns	0.8.0	CoreDNS is a DNS server that chains plugins and...
stable/etcd-operator	0.6.2	CoreOS etcd-operator Helm chart for Kubernetes
stable/external-dns	0.4.4	Configure external DNS servers (AWS Route53...
stable/keel	0.2.0	Open source, tool for automating Kubernetes dep...
stable/msoms	0.1.1	A chart for deploying omsagent as a daemonset...
stable/nginx-lego	0.3.0	Chart for nginx-ingresscontroller and kube-lego
stable/openvpn	2.0.2	A Helm chart to install an openvpn server insid...
stable/risk-advisor	2.0.0	Risk Advisor add-on module for Kubernetes
stable/searchlight	0.1.0	Searchlight by AppsCode – Alerts for Kubernetes
stable/spartakus	1.1.3	Collect information about Kubernetes clusters t...
stable/stash	0.2.0	Stash by AppsCode - Backup your Kubernetes Volumes
stable/traefik	1.15.2	A Traefik based Kubernetes ingress controller w...
stable/voyager	2.0.0	Voyager by AppsCode – Secure Ingress Controller...
stable/weave-cloud	0.1.2	Weave Cloud is a add-on to Kubernetes which pro...
stable/zetcd	0.1.4	CoreOS zetcd Helm chart for Kubernetes
stable/buildkite	0.2.0	Agent for Buildkite

Попробуем другой поиск:

```
> helm search mysql
```

NAME	VERSION	DESCRIPTION
stable/mysql	0.3.4	Fast, reliable, scalable, and easy to use open-...
stable/percona	0.3.0	free, fully compatible, enhanced, open source d...
stable/gcloud-sqlproxy	0.2.2	Google Cloud SQL Proxy
stable/mariadb	2.1.3	Fast, reliable, scalable, and easy to use open-...

Что случилось? Почему mariadb появляется в результатах? Дело в том, что в описании mariadb (ответвление MySQL) упоминается название MySQL, хотя в сокращенном выводе этого не видно. Чтобы получить полное описание, используйте команду `helm inspect`:

```
> helm inspect stable/mariadb
appVersion: 10.1.30
description: Fast, reliable, scalable, and easy to use open-source
  relational database system. MariaDB Server is intended for mission-critical,
  heavy-load production systems as well as for embedding into mass-deployed
  software.
engine: gotpl
home: https://mariadb.org
icon:
https://bitnami.com/assets/stacks/mariadb/img/mariadb-stack-220x234.png
keywords:
- mariadb
- mysql
- database
- sql
- prometheus
maintainers:
- email: containers@bitnami.com
  name: bitnami-bot
name: mariadb
sources:
- https://github.com/bitnami/bitnami-docker-mariadb
- https://github.com/prometheus/mysqld_exporter
version: 2.1.3
```

Установка пакетов

Итак, вы нашли пакет своей мечты. Теперь, вероятно, захотите установить его на свой кластер Kubernetes. Когда вы устанавливаете пакет, Helm создает релиз, который можно применить для отслеживания хода установки. Установим MariaDB, введя команду установки `helm install`. И подробно рассмотрим вывод. В первой

части вывода указаны имя выпуска — `cranky-whippet` в этом случае (можете выбрать собственное имя с помощью флага `-name`), пространство имен и состояние развертывания:

```
> helm install stable/mariadb
NAME:      cranky-whippet
LAST DEPLOYED: Sat Mar 17 10:21:21 2018
NAMESPACE: default
STATUS:    DEPLOYED
```

Во второй части вывода перечислены все ресурсы, созданные этой схемой. Обратите внимание на то, что имена ресурсов являются производными от имени выпуска:

```
RESOURCES:
→ v1/Service
NAME                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
cranky-whippet-mariadb ClusterIP      10.106.206.108   <none>           3306/TCP         1s
→ v1beta1/Deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
cranky-whippet-mariadb 1         1         1             0           1s
→ v1/Pod(related)
NAME                READY     STATUS    RESTARTS   AGE
cranky-whippet-mariadb-6c85fb4796-mttf7 0/1       Init:0/1   0           1s
→ v1/Secret
NAME                TYPE      DATA   AGE
cranky-whippet-mariadb Opaque    2       1s
→ v1/ConfigMap
NAME                DATA   AGE
cranky-whippet-mariadb 1       1s
cranky-whippet-mariadb-tests 1       1s
→ v1/PersistentVolumeClaim
NAME                STATUS   VOLUME
CAPACITY  ACCESS MODES  STORAGECLASS AGE
cranky-whippet-mariadb Bound pvc-9cb7e176-2a07-11e8-9bd6-080027c94384
8Gi       RWO           standard 1s
```

Последняя часть — это заметки, которые дают простые для понимания инструкции по использованию MariaDB в контексте вашего кластера Kubernetes:

```
NOTES:
MariaDB can be accessed via port 3306 on the following DNS name from within
your cluster:
cranky-whippet-mariadb.default.svc.cluster.local
To get the root password run:
MARIADB_ROOT_PASSWORD=$(kubectl get secret --namespace default crankywhippet-
mariadb -o jsonpath="{.data.mariadb-root-password}" | base64 -
decode)
To connect to your database:
1. Run a pod that you can use as a client:
kubectl run cranky-whippet-mariadb-client --rm --tty --image bitnami/mariadb --
MARIADB_ROOT_PASSWORD=$MARIADB_ROOT_PASSWORD --image bitnami/mariadb --
command -- bash
2. Connect using the mysql cli, then provide your password:
mysql -h cranky-whippet-mariadb -p$MARIADB_ROOT_PASSWORD
```

Проверка состояния установки

Helm не ждет завершения установки, потому что это может занять некоторое время. Команда `helm status` отображает последнюю информацию о релизе в том же формате, что и вывод исходной команды установки `helm install`. В выводе команды `install` вы видите, что запрос тома `PersistentVolumeClaim` имел статус `PENDING`. Проверим:

```
> helm status cranky-whippet | grep Persist -A 3
→ v1/PersistentVolumeClaim
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
cranky-whippet-mariadbBoundpvc-9cb7e176-2a07-11e8-9bd6-080027c943848Gi
RWO standard 5m
```

Ура! Он привязан, и теперь у нас есть подключенный том емкостью 8 Гбайт.

Попробуем подключиться и убедиться, что `mariadb` действительно доступна. Немного изменим предлагаемые команды из заметок для подключения. Вместо последовательного запуска `bash` и `mysql` мы можем запустить команду `mysql` прямо в контейнере:

```
> kubectl run cranky-whippet-mariadb-client -rm -tty -I --image
bitnami/mariadb --command -- mysql -h cranky-whippet-mariadb
```

Если вы не видите приглашение командной строки, попробуйте нажать `Enter`:

```
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
+-----+
3 rows in set (0.00 sec)
```

Настройка схемы

Очень часто требуется настроить или сконфигурировать установленные схемы. Helm полностью поддерживает настройку через файлы конфигурации. Чтобы узнать о доступных настройках, снова используйте команду `helm inspect`, но на этот раз сосредоточьтесь на значениях. Вот частичный вывод:

```
> helm inspect values stable/mariadb
## Bitnami MariaDB image version
## ref: https://hub.docker.com/r/bitnami/mariadb/tags/
##
## Default: none
image: bitnami/mariadb:10.1.30-r1
## Specify an imagePullPolicy (Required)
## It's recommended to change this to 'Always' if the image tag is 'latest'
## ref: http://kubernetes.io/docs/user-guide/images/#updating-images
imagePullPolicy: IfNotPresent
## Use password authentication
```

```

usePassword: true
## Specify password for root user
## Defaults to a random 10-character alphanumeric string if not set and
## usePassword is true
## ref:
https://github.com/bitnami/bitnami-docker-mariadb/blob/master/
README.md#setting-the-root-password-on-first-run
##
# mariadbRootPassword:
## Create a database user
## Password defaults to a random 10-character alphanumeric string if not
## set and usePassword is true
## ref:
https://github.com/bitnami/bitnami-docker-mariadb/blob/master/
README.md#creating-a-database-user-on-first-run
##
# mariadbUser:
# mariadbPassword:
## Create a database
## ref:
https://github.com/bitnami/bitnami-docker-mariadb/blob/master/
README.md#creating-a-database-on-first-run
##
# mariadbDatabase:

```

Например, если вы хотите задать пароль администратора и создать базу данных при установке mariadb, то можете создать следующий файл YAML и сохранить его как mariadb-config.yaml:

```

mariadbRootPassword: supersecret
mariadbDatabase: awesome_stuff

```

Затем запустите `helm` и передайте ему файл `yaml`:

```
> helm install -f conf-g.yaml stable/mariadb
```

Вы также можете установить отдельные значения в командной строке с помощью `--set`. Если оба параметра, `--f` и `--set`, пытаются установить одинаковые значения, то `--set` имеет приоритет. Например, в этом случае пароль администратора будет `evenbettersecret`:

```

helm install -f conf-g.yaml --set m-iadbRootPassword=evenbettersecret
stable/mariadb

```

Вы можете указать несколько значений, разделяя их запятыми: `--set a=1, b=2`.

Дополнительные параметры установки

Команда `helm install` поддерживает несколько источников установки:

- ☐ chart repository (как мы видели);
- ☐ локальный архив схем (`helm install foo-0.1.1.tgz`);

- ❑ незапакованный каталог схем (`helm install path/to/foo`);
- ❑ полный URL-адрес (`helm install https://example.com/charts/foo-1.2.3.tgz`).

Обновление и откат релиза

Возможно, вам захочется обновить установленный пакет до последней, самой свежей версии. Helm предоставляет «умную» команду обновления `upgrade`, которая обновляет только то, что было изменено. Например, проверим текущие значения установки `mariadb`:

```
> helm get values cranky-whippet
mariadbDatabase: awesome_stuff
mariadbRootPassword: evenbettersecret
```

Теперь запустим и обновим базу данных, изменив ее имя:

```
> helm upgrade cranky-whippet --set mariadbDatabase=awesome_sauce
stable/mariadb
$ helm get values cranky-whippet
mariadbDatabase: awesome_sauce
```

Обратите внимание на то, что мы потеряли пароль администратора. При обновлении все существующие значения заменяются. Хорошо, давайте откатимся. Команда `helm history` показывает все доступные изменения, от которых можно отказаться:

```
> helm history cranky-whippet
```

REVISION	STATUS	CHART	DESCRIPTION
1	SUPERSEDED	mariadb-2.1.3	Install complete
2	SUPERSEDED	mariadb-2.1.3	Upgrade complete
3	SUPERSEDED	mariadb-2.1.3	Upgrade complete
4	DEPLOYED	mariadb-2.1.3	Upgrade complete

Вернемся к ревизии 3:

```
> helm rollback cranky-whippet 3
Rollback was a success! Happy Helming!
```

```
> helm history cranky-whippet
```

REVISION	STATUS	CHART	DESCRIPTION
1	SUPERSEDED	mariadb-2.1.3	Install complete
2	SUPERSEDED	mariadb-2.1.3	Upgrade complete
3	SUPERSEDED	mariadb-2.1.3	Upgrade complete
4	SUPERSEDED	mariadb-2.1.3	Upgrade complete
5	DEPLOYED	mariadb-2.1.3	Rollback to 3

Проверим, что изменения были отменены:

```
> helm get values cranky-whippet
mariadbDatabase: awesome_stuff
mariadbRootPassword: evenbettersecret
```

Удаление релиза

Разумеется, вы также можете удалить выпуск с помощью команды удаления `helm delete`.

Сначала рассмотрим список релизов. У нас есть лишь один пакет, `cranky-whippet`:

```
> helm list
REVISION      STATUS      CHART          DESCRIPTION
cranky-whippet 5    DEPLOYED      mariadb-2.1.3  default
```

Удалим его:

```
> helm delete cranky-whippet
release "cranky-whippet" deleted
```

Релизов больше нет:

```
> helm list
```

Однако Helm отслеживает и удаленные релизы. Увидеть их можно, используя флаг `--all`:

```
> helm list --all
REVISION      STATUS      CHART          DESCRIPTION
cranky-whippet 5    DELETED      mariadb-2.1.3  default
```

Чтобы удалить релиз полностью, добавьте флаг `--purge`:

```
> helm delete --purge cranky-whippet
```

Работа с репозиториями

Helm хранит схемы в репозиториях, которые являются обычными HTTP-серверами. Любой стандартный HTTP-сервер способен хранить репозиторий Helm. Что касается облачных решений, то команда Helm подтвердила, что AWS S3 и хранилище Google Cloud могут служить репозиториями Helm в веб-режиме. В состав Helm также входит локальный сервер пакетов для тестирования в процессе разработки. Он работает на клиентском компьютере, поэтому не подходит для совместного применения. Если команда небольшая, можете запустить сервер пакетов Helm на общем компьютере, доступном для всех участников локальной сети.

Чтобы использовать локальный сервер пакетов, введите команду `helm serve`. Она блокирующая, поэтому запускайте ее в отдельном окне терминала. По умолчанию Helm начнет раздавать схемы из каталога `~/helm/repository/local`. Можете разместить там свои схемы и создать индексный файл с помощью команды `helm index`.

В сгенерированном файле `index.yaml` перечислены все схемы.

Обратите внимание на то, что Helm не предоставляет инструменты для загрузки схем в удаленные репозитории, поскольку в этом случае удаленный сервер должен быть совместим с Helm, иначе он не будет знать, куда положить схему и как обновить файл `index.yaml`.

На стороне клиента команда `helm repo` поддерживает следующие действия: `list`, `add`, `remove`, `index` и `update`:

> `helm repo`

Эта команда состоит из нескольких подкоманд для взаимодействия с репозиториями схем. Ее можно применить для добавления, удаления, вывода содержимого и индексации репозитория схем.

❑ Пример использования:

```
$ helm repo add [ИМЯ] [URL_РЕПОЗИТОРИЯ]
```

❑ Использование:

```
helm repo [команда]
```

❑ Доступные команды:

<code>add</code>	добавляет репозиторий <code>chart</code>
<code>index</code>	генерирует индексный файл для данного каталога
<code>list</code>	выводит список репозитория <code>chart</code>
<code>remove</code>	удаляет репозиторий <code>chart</code>
<code>update</code>	обновляет данные доступных репозитория <code>charts</code>

Управление схемами с помощью Helm

Helm предоставляет несколько команд для управления схемами. Он может создать для вас новую схему:

```
> helm create cool-chart
Creating cool-chart
```

Helm создаст следующие файлы и каталоги внутри `cool-chart`:

```
-rw-r--r-- 1 gigi.sayfan gigi.sayfan 333B Mar 17 13:36 .helmignore
-rw-r--r-- 1 gigi.sayfan gigi.sayfan 88B Mar 17 13:36 Chart.yaml
drwxr-xr-x 2 gigi.sayfan gigi.sayfan 68B Mar 17 13:36 charts
drwxr-xr-x 7 gigi.sayfan gigi.sayfan 238B Mar 17 13:36 templates
-rw-r--r-- 1 gigi.sayfan gigi.sayfan 1.1K Mar 17 13:36 values.yaml
```

После того как вы отредактировали схему, можете упаковать ее в сжатый архив `tar`:

```
> helm package cool-chart
```

Helm создаст архив с именем `cool-chart-0.1.0.tgz` и сохранит его как в локальном каталоге, так и в локальном репозитории.

Вы также можете использовать `helm`, чтобы найти проблемы с форматированием или информацией своей схемы:

```
> helm lint cool-chart
==> Linting cool-chart
[INFO] Chart.yaml: icon is recommended
1 chart(s) linted, no failures
```

Преимущества стартовых пакетов. Команда `helm create` поддерживает необязательный флаг `--starter`, который позволяет указать стартовую схему.

Стартеры — это регулярные схемы, расположенные в `$HELM_HOME/starters`. Разработчики могут создавать схемы, специально предназначенные для использования в качестве стартеров. Они должны быть разработаны с учетом следующих соображений.

- ❑ `Chart.yaml` будет перезаписан генератором.
- ❑ Пользователи рассчитывают на то, что содержимое схемы можно изменять, поэтому данный процесс должен быть описан в документации.

Сейчас невозможно установить схемы в `$HELM_HOME/starters`, пользователь должен скопировать их вручную. Если разрабатываете схемы стартового пакета, не забудьте упомянуть об этом в их документации.

Создание своих собственных схем

Схема представляет собой набор файлов, которые описывают связанные между собой ресурсы Kubernetes. Одну и ту же схему можно использовать для развертывания чего-то простого, например пода `memcached`, или чего-то сложного, допустим полного стека веб-приложений с HTTP-серверами, базами данных и кэшами.

Схемы создаются в виде файлов, размещенных в определенном дереве каталогов. Затем они могут быть упакованы в архивы с поддержкой версий, готовые к развертыванию. Ключевой файл — `Chart.yaml`.

Файл Chart.yaml

`Chart.yaml` — основной файл схемы Helm. Обязательными полями в нем являются `name` и `version`:

- ❑ `name` — название схемы (такое же, как и у каталога);
- ❑ `version` — версия в формате SemVer 2.

Он также может содержать необязательные поля:

- ❑ `kubeVersion` — диапазон совместимых версий Kubernetes в формате SemVer;
- ❑ `description` — описание проекта одним предложением;
- ❑ `keywords` — список ключевых слов для проекта;
- ❑ `home` — URL-адрес домашней страницы проекта;
- ❑ `sources` — список URL-адресов с исходными кодами для проекта;
- ❑ `maintainers`:
 - `name` — имя мейнтейнера (требуется для каждого мейнтейнера);
 - `email` — электронная почта мейнтейнера (не обязательно);
 - `url` — URL-адрес мейнтейнера (не обязательно);

- ❑ `engine` — имя движка шаблона (по умолчанию `getpl`);
- ❑ `icon` — URL изображения в формате SVG или PNG, которое будет использоваться как значок;
- ❑ `appVersion` — версия приложения;
- ❑ `deprecated` — логическое значение, показывающее, устарела ли эта схема;
- ❑ `tillerVersion` — версия Tiller, которая требуется схеме.

Версионность схем

Поле версии в файле `Chart.yaml` используется клиентом командной строки и сервером Tiller. Команда `helm package` применит версию, которую найдет в файле `Chart.yaml` при построении имени пакета. Номер версии в имени пакета схемы должен соответствовать номеру версии в файле `Chart.yaml`.

Поле `appVersion`

Поле `appVersion` не связано с полем версии. Оно не используется Helm и служит метаданными или документацией для пользователей, которые хотят понять, что именно они развертывают. Helm не следит за его корректностью.

Устаревшие схемы

Время от времени некоторые схемы устаревают. Отметьте схему как устаревшую, установив для поля `deprecated` в файле `Chart.yaml` значение `true`. Этого достаточно, чтобы обозначить последнюю версию схемы как устаревшую. Позже под этим именем можно опубликовать более новую версию. Рабочий процесс, используемый проектом `kubernetes/charts`, таков.

- ❑ Обновление файла схемы `Chart.yaml`, чтобы пометить ее как устаревшую и повысить версию.
- ❑ Выпуск новой версии схемы.
- ❑ Удаление схемы из исходного репозитория.

Файлы метаданных схемы

Схемы могут содержать различные файлы метаданных, например `README.md`, `LICENSE` и `NOTES.txt`, которые описывают установку, настройку, использование и лицензию схемы. Файл `README.md` должен быть в формате Markdown и предоставлять следующую информацию:

- ❑ описание приложения или сервиса, которые предоставляет схема;
- ❑ любые требования для установки и запуска схемы;
- ❑ описание параметров в `values.yaml` и значения по умолчанию;
- ❑ любые другие сведения, которые могут иметь отношение к установке или настройке схемы.

Файл `templates/NOTES.txt` будет отображаться после установки или просмотра статуса релиза. Вы должны сделать файл `NOTES` кратким и сослаться в нем на файл `README.md` для подробных объяснений. Обычно используются примечания о применении и последующие шаги, такие как информация о подключении к базе данных или доступе к пользовательскому веб-интерфейсу.

Управление зависимостями схемы

В Helm схема может зависеть от любого количества других схем. Эти зависимости обозначают явно, перечисляя их в файле `requirements.yaml` или копируя зависящие схемы в подкаталог `charts/` во время установки.

Зависимость может быть либо архивом схемы (`foo-1.2.3.tgz`), либо ее распакованным каталогом. Однако ее имя не должно начинаться с символа подчеркивания или точки. Такие файлы игнорируются загрузчиком схем.

Управление зависимостями с `requirements.yaml`

Вместо того чтобы вручную размещать схемы в подкаталоге `chart/`, лучше объявить зависимости, используя файл `requirements.yaml` внутри своей схемы.

`requirements.yaml` — это простой файл для перечисления зависимостей схемы:

```
dependencies:
- name: foo
  version: 1.2.3
  repository: http://example.com/charts
- name: bar
  version: 4.5.6
  repository: http://another.example.com/charts
```

Поле `name` — это название нужной схемы.

Поле `version` — это версия нужной схемы.

Поле `repository` — полный URL `chart repository`. Обратите внимание: для локального добавления этого репозитория вы должны использовать команду `helm repo`.

Создав файл зависимостей, вы можете выполнить команду `helm dep up`, и она загрузит в подкаталог `charts` все указанные в нем схемы:

```
$ helm dep up foo-chart
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "example" chart repository
...Successfully got an update from the "another" chart repository
Update Complete. Happy Helming!
Saving 2 charts
Downloading Foo from repo http://example.com/charts
Downloading Bar from repo http://another.example.com/charts
```

Helm хранит схемы зависимостей, извлекаемые во время выполнения команды `helm dep up`, в виде архивов схем в каталоге `chart/`. В предыдущем примере в каталоге `charts` будут находиться следующие файлы:

```
charts/
  foo-1.2.3.tgz
  bar-4.5.6.tgz
```

Управление схемами и их зависимостями с помощью файла `requirements.yaml` — наилучший подход как для явного документирования зависимостей и совместной работы с ними в команде, так и для поддержки автоматизированных конвейеров.

Использование специальных полей в файле `requirements.yaml`

Каждая запись в файле `requirements.yaml` может содержать необязательные теги `fields` и условия.

Эти поля могут применяться для динамического управления загрузкой схем (по умолчанию загружаются все схемы). Когда присутствуют теги или условия, Helm будет их оценивать и определять, должна ли загружаться целевая схема.

- ❑ **condition.** Поле `condition` содержит один или несколько путей к файлам YAML, разделенных запятыми. Если этот путь существует среди значений самого верхнего родителя и указывает на булево значения, схема будет включена или отключена на основе этого значения. Оценивается только первый допустимый путь, найденный в списке, а если путей нет, условие не действует.
- ❑ **tags.** Поле `tags` представляет собой список меток в формате YAML для сопоставления с этой схемой. В значениях самого верхнего родителя все схемы с тегами можно включить или отключить, указав тег и булево значение.

Далее приведены примеры файлов `requirements.yaml` и `values.yaml`, которые используют условия и теги для включения и отключения установки зависимостей. Файл `requirements.yaml` определяет два условия для установки его зависимостей на основе значения поля `global.enabled` и специального поля `sub-charts.enabled`:

```
# parentchart/requirements.yaml
dependencies:
  - name: subchart1
    repository: http://localhost:10191
    version: 0.1.0
    condition: subchart1.enabled, global.subchart1.enabled
    tags:
      - front-end
      - subchart1
  - name: subchart2
    repository: http://localhost:10191
```

```

version: 0.1.0
condition: subchart2.enabled,global.subchart2.enabled
tags:
  - back-end
  - subchart2

```

Файл `values.yaml` присваивает значения некоторым условным переменным. У тега `subchart2` нет значения, поэтому он считается включенным:

```

# parentchart/values.yaml
subchart1:
  enabled: true
tags:
  front-end: false
  back-end: true

```

Вы также можете установить значения тегов и условий из командной строки при установке схемы, и они получат приоритет перед файлом `values.yaml`:

```
helm install --set subchart2.enabled=false
```

Интерпретация тегов и условий подчиняется следующим правилам.

- ❑ Условия (установленные в виде значений) всегда переопределяют теги. Выигрывает первое из всех условных ответвлений, а последующие ответвления для этой схемы игнорируются.
- ❑ Схема включается, если какой-либо из ее тегов истинный.
- ❑ Значения тегов и условий должны присутствовать среди значений самого верхнего родителя.
- ❑ Теги: значения ключа должны быть ключами самого верхнего уровня. Глобальные значения и вложенные теги не поддерживаются.

Использование шаблонов и значений

Любое важное приложение приходится настраивать и адаптировать к конкретному сценарию использования. Схемы Helm — это шаблоны языка Go, которые заполняют подстановочные выражения. Helm поддерживает дополнительные функции из библиотеки Spring и еще несколько специализированных функций. Файлы шаблонов хранятся в подкаталоге `templates/` схемы. Helm использует механизм шаблонов для отображения всех файлов в этом каталоге и применения предоставленных файлов со значениями.

Создание файлов шаблонов

Файлы шаблонов — это обычные текстовые файлы, которые соответствуют правилам языка шаблонов Go. Они могут генерировать файлы конфигурации Kubernetes.

Вот служебный шаблонный файл из схемы `artifactory`:

```
kind: Service
apiVersion: v1
kind: Service
metadata:
  name: {{ template "artifactory.fullname" . }}
  labels:
    app: {{ template "artifactory.name" . }}
    chart: {{ .Chart.Name }}-{{ .Chart.Version }}
    component: "{{ .Values.artifactory.name }}"
    heritage: {{ .Release.Service }}
    release: {{ .Release.Name }}
{{- if .Values.artifactory.service.annotations }}
  annotations:
    {{ toYaml .Values.artifactory.service.annotations | indent 4 }}
{{- end }}
spec:
  type: {{ .Values.artifactory.service.type }}
  ports:
    - port: {{ .Values.artifactory.externalPort }}
      targetPort: {{ .Values.artifactory.internalPort }}
      protocol: TCP
      name: {{ .Release.Name }}
  selector:
    app: {{ template "artifactory.name" . }}
    component: "{{ .Values.artifactory.name }}"
    release: {{ .Release.Name }}
```

Использование конвейеров и функций. Helm позволяет использовать в файлах шаблонов развитый и сложный синтаксис, используя встроенные шаблонные функции языка Go и Spring, а также конвейеры. Вот пример шаблона, который демонстрирует эти возможности. Он использует функции `repeat`, `quote` и `upper` для ключей `food` и `drink`, а также конвейеры — для соединения нескольких функций:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  greeting: "Hello World"
  drink: {{ .Values.favorite.drink | repeat 3 | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
```

Посмотрим, есть ли в файле значений следующий раздел:

```
favorite:
  drink: coffee
  food: pizza
```

Если да, то итоговая схема будет такой:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cool-app-configmap
data:
  greeting: "Hello World"
  drink: "coffeecoffeecoffee"
  food: "PIZZA"
```

Встраивание предопределенных значений

Helm предоставляет некоторые предопределенные значения, которые вы можете использовать в своих шаблонах. В предыдущем шаблоне схемы `artifactory` такими значениями были `Release.Name`, `Release.Service`, `Chart.Name` и `Chart.Version` — это примеры предопределенных значений Helm. Среди других предопределенных значений можно выделить следующие:

- ☐ `Release.Time`;
- ☐ `Release.Namespace`;
- ☐ `Release.IsUpgrade`;
- ☐ `Release.IsInstall`;
- ☐ `Release.Revision`;
- ☐ `Chart`;
- ☐ `Files`;
- ☐ `Capabilities`.

Схема — это содержимое файла `Chart.yaml`. Предопределенные значения в файлах — это объекты вида «ключ — значение», которые обеспечивают доступ через различные функции. Обратите внимание на то, что неизвестные поля в поле `Chart.yaml` игнорируются шаблонизатором и не могут использоваться для передачи в шаблоны произвольно структурированных данных.

Подача значений из файла

Вот часть файла со значениями по умолчанию в схеме `artifactory`. Значения из этого файла используются для заполнения нескольких шаблонов. Например, в предыдущем служебном шаблоне применяются значения `artifactory` и `internalPort`:

```
artifactory:
  name: artifactory
  replicaCount: 1
  image:
    # repository: "docker.bintray.io/jfrog/artifactory-oss"
    repository: "docker.bintray.io/jfrog/artifactory-pro"
    version: 5.9.1
```

```
pullPolicy: IfNotPresent
service:
  name: artifactory
  type: ClusterIP
  annotations: {}
  externalPort: 8081
  internalPort: 8081
  persistence:
    mountPath: "/var/opt/jfrog/artifactory"
    enabled: true
    accessMode: ReadWriteOnce
    size: 20Gi
```

Во время выполнения команды установки можно указать собственный файл в формате YAML, который переопределит значения по умолчанию:

```
> helm install --values=custom-values.yaml gitlab-ce
```

Область, зависимости и значения

Файлы значений могут объявлять значения для схем верхнего уровня, а также для любой из схем, которые включены в каталог charts/. Например, файл artifactory-cevalues.yaml содержит некоторые значения по умолчанию для своей схемы зависимостей postgresql:

```
## Configuration values for the postgresql dependency
## ref:
## https://github.com/kubernetes/charts/blob/master/stable/postgresql/README.md
##
postgresql:
  postgresUser: "artifactory"
  postgresPassword: "artifactory"
  postgresDatabase: "artifactory"
  persistence:
    enabled: true
```

Схема верхнего уровня имеет доступ к значениям зависимых схем, но не наоборот. Предусмотрено также глобальное значение, доступное для всех схем. Например, вы можете добавить что-то вроде этого:

```
global:
  app: cool-app
```

Глобальный параметр (если таковой присутствует) будет реплицирован между значениями всех зависимых схем, как показано далее:

```
global:
  app: cool-app
postgresql:
  global:
    app: cool-app
  ...
```

Резюме

В этой главе мы рассмотрели Helm — диспетчер пакетов Kubernetes. С помощью Helm система Kubernetes управляет сложным программным обеспечением, состоящим из множества взаимозависимых ресурсов. Он выполняет те же задачи, что и менеджер пакетов в операционной системе. Helm организует пакеты и позволяет вам искать схемы, устанавливать и обновлять их, а также делиться схемами с коллегами. Вы можете создавать свои схемы и хранить их в репозиториях.

На этом этапе вы должны понимать, насколько важную роль играет Helm в экосистеме и сообществе Kubernetes. Вы должны иметь возможность продуктивно использовать его и даже разрабатывать собственные схемы и делиться ими.

В следующей главе мы поговорим о будущем Kubernetes и рассмотрим его «дорожную карту». А вдобавок — несколько пунктов из моего личного списка пожеланий.

14

Будущее Kubernetes

В этой главе мы рассмотрим перспективы развития Kubernetes с разных точек зрения. Начнем с «дорожной карты» и новых функций продукта, а затем погрузимся в процесс его проектирования. Вспомним, как система Kubernetes зарождалась, поговорим о сообществе, экосистеме и осведомленности о ней потребителей. Бóльшая часть нового функционала Kubernetes станет определяться тем, как это поможет системе в противостоянии с конкурентами. Просветительская работа также будет играть важную роль, поскольку контейнерная оркестрация — это новая, быстро развивающаяся и пока еще не всем понятная область. Затем мы обсудим динамические дополнения.

Итак, мы рассмотрим следующие темы.

- ❑ Дорога в будущее.
- ❑ Конкуренция.
- ❑ Время Kubernetes.
- ❑ Просвещение и обучение.
- ❑ Модулирование и дополнения «вне дерева».
- ❑ Технология service mesh и serverless-фреймворки.

Дорога в будущее

Kubernetes — это крупный проект с открытым исходным кодом. Рассмотрим некоторые из запланированных функций и будущих релизов, а также различные специальные группы по интересам, которые сосредоточены на конкретных областях.

Версии и этапы развития Kubernetes

Kubernetes обновляется довольно регулярно. Текущий выпуск по состоянию на апрель 2018 года — 1.10. Приведу несколько выдержек из релизов 1.11, чтобы дать представление о проделанной работе.

- ❑ Обновление до Go 1.10.1 и переход на сервер etcd версии 3.2.
- ❑ Поддержка сторонних поставщиков проверки подлинности.
- ❑ Перенос kublet-флагов в `kublet.config.k8s.io`.

- ❑ Поддержка стандартного балансировщика нагрузки Azure и публичного IP-адреса.
- ❑ Добавление команды `kubectl api-resources`.
- ❑ Минорные релизы выпускаются каждые три месяца, а патчи закрывают уязвимости и недоработки до следующего минорного выпуска. Далее приведены даты трех последних выпусков:
 - 10.0 — 26 марта 2018 года, 1.9.6 — 21 марта 2018 года;
 - 9.0 — 15 декабря 2017 года, 1.8.5 — 7 декабря 2017 года;
 - 8.0 и 1.7.7 — 28 сентября 2017 года (мой день рождения!).

Еще один хороший способ понять, что происходит, — посмотреть на изменения в альфа- и бета-версиях. Вы можете просмотреть журнал изменений по адресу github.com/kubernetes/kubernetes/blob/master/CHANGELOG.md.

Вот некоторые из основных тем выпуска 1.10:

- ❑ Node;
- ❑ Network;
- ❑ Storage;
- ❑ Windows;
- ❑ OpenStack;
- ❑ API machinery;
- ❑ Auth;
- ❑ Azure;
- ❑ CLI.

Особые интересы и рабочие группы Kubernetes

Большая часть работы по созданию Kubernetes — крупного проекта с открытым исходным кодом — протекает в нескольких рабочих группах сообщества. Полный список здесь: github.com/kubernetes/community/blob/master/sig-list.md.

Планируются будущие релизы в основном в рамках этих SIG и рабочих групп, потому что система Kubernetes слишком велика, чтобы обрабатывать все централизованно. Группы регулярно встречаются и обсуждают проект.

Конкуренция

В мае 2017 года было опубликовано первое издание этой книги. В то время проект Kubernetes существовал в совершенно иной конкурентной среде. Вот что я написал тогда:

«Kubernetes работает в одной из самых горячих технологических областей контейнерной оркестрации. Будущее Kubernetes должно рассматриваться как часть всего рынка. Как вы увидите, некоторые из вероятных конкурентов могут быть и партнерами, которые рекламируют как собственное предложение, так и Kubernetes (или, по крайней мере, Kubernetes может работать на их платформе)».

Менее чем через год ситуация резко изменилась: Kubernetes победила. Все провайдеры облачных услуг предлагают управляемые услуги Kubernetes. IBM обеспечивает ее поддержку на «голых» кластерах. Компании, которые разрабатывают программное обеспечение и дополнения для контейнерной оркестрации, сосредоточены на Kubernetes, а не на создании продуктов, поддерживающих несколько решений оркестрации.

Значимость контейнеризации

Контейнерные платформы для оркестрации, такие как Kubernetes, прямо и косвенно конкурируют с большими и меньшими областями. Например, Kubernetes может быть доступна на облачной платформе, такой как AWS, но не подойдет для безусловного перехода на такую платформу. В то же время Kubernetes лежит в основе GKE на Google Cloud Platform. Разработчики, выбирающие более высокий уровень абстракции, допустим облачную платформу или даже PaaS, чаще всего будут применять решение по умолчанию. Но некоторые разработчики или организации беспокоятся о том, чтобы не пострадать из-за блокировки провайдера, или должны работать на нескольких облачных или гибридных публичных/частных платформах. Здесь у Kubernetes есть весомое преимущество. Хотя объединение казалось потенциальной серьезной угрозой для принятия Kubernetes, теперь каждый крупный игрок предлагает эту систему прямо на своей платформе или в решении.

Docker Swarm

Docker сейчас стандарт де-факто для контейнеров (хотя и CoreOS rkt набирает обороты). Docker хочет получить кусок оркестрового пирога, поэтому выпустил продукт Docker Swarm. Основное преимущество кластера Docker Swarm заключается в том, что он входит в состав установки Docker и использует стандартные API Docker. Таким образом, кривая обучения оказывается не слишком крутой и начать осваивать эти продукты легче, чем прочие. Тем не менее Docker Swarm отстает от Kubernetes с точки зрения возможностей и готовности. Кроме того, Docker не отличается безупречной репутацией и в том случае, когда речь идет о высококачественной технике и безопасности. Организации и разработчики, заинтересованные

в стабильности своих систем, могут отказаться от Docker Swarm. Разработчики Docker осознают проблему и предпринимают шаги для ее устранения. Они выпустили продукт Enterprise, а также переработали внутренние элементы Docker как набор независимых компонентов через проект Moby. Но недавно и разработчики Docker признали высокое место Kubernetes как платформы для оркестрации контейнеров. Я предполагаю, что Docker Swarm выдохнется и будет использоваться только для очень маленьких прототипов.

Mesos/Mesosphere

Mesosphere — компания, стоящая за открытым исходным кодом Apache Mesos, а продукт DC/OS — это платформа, отвечающая за управление контейнерами и большими данными в облаке. Технология устоявшаяся, и Mesosphere развивает ее, но у них нет ресурсов и импульса, которыми обладает Kubernetes. Я считаю, что Mesosphere будет очень стабильно развиваться, потому что это большой рынок, но не сможет угрожать Kubernetes как решению для оркестрации номер один. Кроме того, Mesosphere также признала, что они не могут победить Kubernetes и решили присоединиться к ней. В DC/OS 1.11 вы получаете Kubernetes как сервис. Предложение DC/OS — доступное, очень простое в настройке и безопасное по умолчанию развертывание Kubernetes, которое было протестировано в Google, AWS и Azure.

Облачные платформы

Множество организаций и разработчиков переходят на публичные облачные платформы, чтобы избежать головной боли из-за низкоуровневого управления своей инфраструктурой. Основная мотивация этих компаний заключается в том, чтобы быстро продвигаться в работе и сосредоточиться на своей основной задаче. Таким образом, они наверняка станут использовать решение для развертывания по умолчанию, предлагаемое их облачным провайдером, потому что интеграция — это наиболее безболезненное и оптимизированное решение.

AWS

Kubernetes отлично работает на AWS через официальный проект Kubernetes Kops: github.com/kubernetes/kops.

Некоторые функции Kops:

- ☐ автоматизация выделения кластеров Kubernetes в AWS;
- ☐ развертывание высокодоступных мастеров Kubernetes;
- ☐ возможность генерации конфигураций Terraform.

Однако Kops — это не официальное решение AWS. Если вы управляете своей инфраструктурой через консоль AWS и API, то путь наименьшего сопротивления — *Elastic Container Service (ECS)* — встроенное решение для организации оркестрации, не основанное на Kubernetes.

Теперь AWS полностью предана идее Kubernetes и в настоящее время занимается выпуском *Elastic Kubernetes Service (EKS)* — полностью управляемого и высокодоступного кластера Kubernetes без каких-либо изменений, но тесно интегрированного с помощью надстроек и дополнений в сервисы AWS.

В первом издании я предположил, что могущественный AWS будет держиваться своей политики и стоять за ECS, но ошибся — даже он откатился к Kubernetes. ECS будет поддерживаться, потому что многие организации инвестировали в него и, возможно, не захотят мигрировать на Kubernetes. Однако я предсказываю, что со временем ECS будет признан устаревшим продуктом, поддерживаемым для организаций, не имеющих достаточных стимулов для перехода на Kubernetes.

Azure

Azure предоставляет услугу контейнера Azure, но не навязывает фаворитов. Вы можете выбрать, хотите ли использовать Kubernetes, Docker Swarm или DC/OS. Это удобно, потому что изначально Azure была основана на Mesosphere DC/OS, а позже добавили Kubernetes и Docker Swarm в качестве вариантов оркестрации. Поскольку Kubernetes значительно опережает прочих по возможностям и готовности, я считаю, что она станет функцией оркестрации номер один на Azure.

Во второй половине 2017 года Azure официально выпустила *Azure Kubernetes Service (AKS)*, и Microsoft полностью уступила Kubernetes первенство в качестве решения для оркестрации контейнеров. Сообщество Kubernetes очень активно, оно приобрело Deis (разработчик Helm) и создает множество инструментов, а также средств интеграции. Поддержка Windows для Kubernetes и интеграция с Azure продолжают улучшаться.

Alibaba Cloud

Alibaba Cloud — это китайский AWS. Их API специально сделан очень похожим на API AWS. Alibaba Cloud использовал сервис управления контейнерами на основе Docker Swarm. Я развернул несколько небольших приложений на Alibaba Cloud и сделал вывод, что разработчики, похоже, не отстают от крупных игроков и быстро вносят изменения вслед за ними. За последний год Alibaba Cloud (Aliyun) вошел в ряды сторонников Kubernetes. Существует несколько ресурсов для развертывания кластеров Kubernetes и управления ими в облаке Alibaba, включая реализацию GitHub интерфейса облачного провайдера.

Время Kubernetes

Kubernetes активно развивается, ее сообщество увеличивается. Пользователи стекаются к Kubernetes по мере того, как растет осведомленность о ней, СМИ признают, что она занимает лидирующую позицию, экосистема развивается, крупные корпорации и компании вслед за Google активно поддерживают ее, а многие оценивают и запускают в производство.

Сообщество

Сообщество разработчиков — один из главных активов Kubernetes. Недавно Kubernetes стала первым проектом, который завершил полный цикл развития в рамках *Cloud Native Computing Foundation (CNCF)*.

GitHub

Kubernetes разрабатывается на GitHub и входит в 0,01 % самых популярных проектов на этой платформе, удерживая первенство по активности. Обратите внимание на то, что за последний год Kubernetes стала более модульной и многие части пазла теперь разрабатываются отдельно.

В своем профиле LinkedIn профессионалы указывают Kubernetes чаще, чем любое другое сопоставимое предложение.

Год назад Kubernetes имела примерно 1100 участников и 34 000 коммитов. Теперь эти цифры возросли до более чем 1600 участников и свыше 63 000 коммитов (рис. 14.1).



Рис. 14.1

Конференции и встречи

Другой признак быстрого развития Kubernetes — количество конференций, встреч и их участников. Количество участников KubeCon быстро растет, и новые встречи, посвященные Kubernetes, организуются каждый день.

Осведомленность потребителей

Kubernetes привлекает пристальное внимание пользователей, и она часто развешивается. Крупные и малые компании, которые попадают на арену контейнеров/DevOps/микросервисов, принимают Kubernetes, и эта тенденция очевидна. Один из интересных показателей — количество новых вопросов на сайте Stack Overflow. Сообщество предпринимает шаги, чтобы отвечать на эти вопросы и способствовать сотрудничеству. По темпам роста Kubernetes намного превосходит своих соперников, и это явно видно (рис. 14.2).

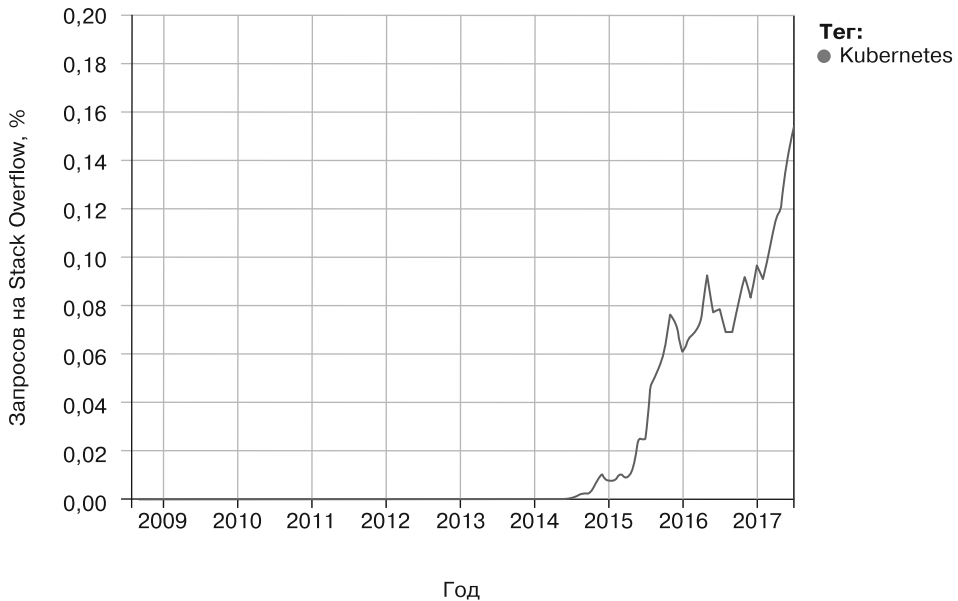


Рис. 14.2

Экосистема

Эволюция Kubernetes очень впечатляет: от облачных провайдеров до платформ PaaS и стартапов, которые предлагают комфортную среду выполнения.

Провайдеры публичных облаков

Все основные поставщики облачных услуг напрямую поддерживают Kubernetes. Очевидным лидером является компания Google с системой GKE, которая представляет собой встроенный механизм контейнеров на базе Google Cloud Platform. Проект Корп, упомянутый ранее, — это хорошо поддерживаемое, обслуживаемое

и документированное решение для AWS, и выпуск EKS уже не за горами. Azure предлагает AKS. В облачном сервисе контейнера IBM работает Kubernetes. Oracle внимательно следит за Kubernetes и предлагает услуги контейнеризации, основанные на новейших версиях Kubernetes и Kubeadm.

OpenShift

OpenShift — это контейнерный прикладной продукт RedHat, построенный на базе открытого исходного кода OpenShift, основанного на Kubernetes. OpenShift добавляет управление жизненным циклом приложения и инструмент DevOps поверх Kubernetes и вносит в нее большой вклад (например, автомасштабирование). Этот тип взаимодействия очень полезный и обнадеживающий. RedHat недавно приобрела CoreOS, и слияние CoreOS Tectonic с OpenShift может обеспечить отличный эффект от совместного использования.

OpenStack

OpenStack — это частная облачная платформа с открытым исходным кодом, и недавно она приняла решение о стандартизации Kubernetes в качестве базовой платформы для оркестрации. Это большое дело, потому что крупные предприятия, которые хотят действовать в совокупности публичных и частных облаков, будут гораздо лучше интегрироваться с облачной федерацией Kubernetes, с одной стороны, и OpenStack — частной облачной платформой, применяющей Kubernetes за кадром, — с другой.

В последнем обзоре OpenStack, выполненном в ноябре 2017 года, показано, что Kubernetes сейчас — это самое популярное решение для контейнерной оркестрации (рис. 14.3).

Другие игроки

Существует и еще ряд компаний, которые используют Kubernetes в качестве основы, например Rancher и Apprenda. Большое количество стартапов разрабатывают надстройки и сервисы, запускающиеся внутри кластера Kubernetes. Ее ждет яркое будущее.

Обучение и подготовка

Просвещение будет иметь решающее значение. На смену энтузиастам, применяющим Kubernetes с самых ранних версий, приходят обычные пользователи, поэтому очень важно, чтобы организации и разработчики имели доступ к подходящим ресурсам; это будет способствовать быстрому и продуктивному освоению данной технологии. Уже существуют неплохие ресурсы, и я предсказываю, что в дальнейшем их количество будет расти, а качество — улучшаться. И конечно, книга, которую вы сейчас читаете, — часть этого процесса.

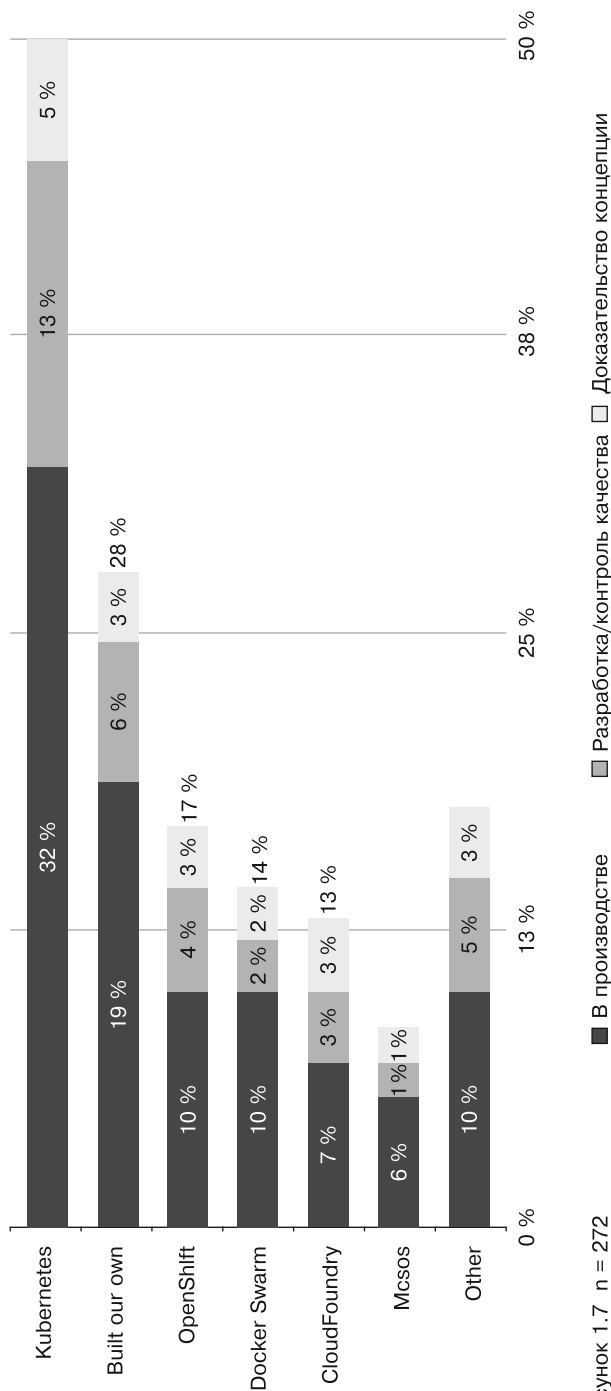


Рисунок 1.7 n = 272

Рис. 14.3

Официальная документация Kubernetes становится все лучше и лучше. Онлайн-учебники отлично подходят для начала работы с этой платформой.

- ❑ В CNCF есть бесплатный вводный курс Kubernetes (а также более продвинутый платный курс): www.cncf.io/certification/training/.
- ❑ Сотрудники Google создали несколько курсов Udacity по Kubernetes. Найдете их по адресу www.udacity.com/course/scalable-microservices-with-kubernetes--ud615.
- ❑ Отличный ресурс — KataCoda, он предоставляет бесплатную игровую площадку Kubernetes, где вы можете получить частный кластер за считанные секунды, а также несколько практических руководств по расширенным темам на www.katacoda.com/courses/kubernetes.

Есть также много платных специализированных учебных курсов по Kubernetes. Поскольку популярность платформы растет, их количество будет только увеличиваться.

Модулирование и дополнения «вне дерева»

Kubernetes с самого первого выпуска добилась больших успехов в области модулирования. Она всегда была образцом гибкости и расширяемости. Однако первоначально вам приходилось создавать код и связывать его с сервером API Kubernetes или с Kublet (за исключением дополнений CNI). Вы также должны были проверять код и интегрировать его с основной базой Kubernetes, чтобы сделать доступным для других разработчиков. В то время я был очень рад динамическим дополнениям Go 1.8 и тому, как их можно было гибко использовать для расширения Kubernetes. Разработчики и сообщество Kubernetes пошли по другому пути и решили сделать ее универсальным движком, где почти каждый аспект реально настроить или расширить извне через стандартные интерфейсы. Вы видели много примеров таких действий в главе 12. Подход «вне дерева» означает, что вы интегрируете дополнение или расширение с Kubernetes, которые находятся вне дерева кода Kubernetes на GitHub. Существует несколько механизмов.

- ❑ Дополнения CNI используют стандартный ввод и вывод через отдельные исполняемые файлы.
- ❑ Дополнения CSI применяют поды gRPC.
- ❑ Дополнения kubectl используют дескрипторы YAML и двоичные команды.
- ❑ Агрегаторы API применяют пользовательские серверы API.
- ❑ Веб-хуки используют удаленные HTTP-интерфейсы.
- ❑ Различные другие дополнения могут быть развернуты как поды.
- ❑ Используются провайдеры внешних удостоверений.

Технология service mesh и serverless-фреймворки

Kubernetes помогает выполнять тяжелую работу, связанную с оркестрацией контейнеров, и сокращать затраты благодаря эффективному планированию. Но в облачном мире есть две набирающие силу тенденции. Технология service mesh и serverless-фреймворки идеально подходят Kubernetes.

Технология service mesh

Технология service mesh работает на более высоком уровне, чем контейнерная оркестрация, и управляет услугами. «Сетки для сервисов» предоставляют различные возможности, крайне необходимые при запуске систем с сотнями и тысячами различных сервисов, таких как:

- ❑ динамическая маршрутизация;
- ❑ латентно-ориентированная балансировка нагрузки между востоком и западом (внутри кластера);
- ❑ автоматические повторы идемпотентных запросов;
- ❑ операционные показатели.

Раньше приложениям приходилось выполнять эти обязанности вдобавок к основным функциям. Теперь «сетки для сервисов» снимают с них нагрузку и обеспечивают такой уровень инфраструктуры, чтобы они могли сосредоточиться на своих основных целях.

Самая известная «сетка для сервисов» — Linkerd компании Buoyant. Linkerd поддерживает Kubernetes, а также других оркестрантов. Но, получив импульс в виде Kubernetes, разработчики из Buoyant решили разработать новую «сетку для сервисов» Kubernetes, названную Conduit (в Rust). Это еще одно свидетельство инновационности Kubernetes. Другая «сетка для сервисов» Kubernetes — Istio. Проект Istio был основан командами из Google, IBM и Lyft. Он построен на базе «сетки для сервисов» Lyft Envoy и быстро развивается.

Serverless-фреймворки

Бессерверные вычисления — поразительная новая тенденция в облачном ландшафте. Функции AWS Lambda сейчас самые популярные, и все облачные платформы их обеспечивают. Их суть состоит в том, что вам не нужно предоставлять оборудование, сетевые узлы и хранилище. Вместо этого вы просто пишете свой код, упаковываете его (часто в контейнер) и вызываете, когда захотите. Облачная платформа заботится о распределении ресурсов для запуска вашего кода во время вызова и освобождения ресурсов при завершении работы. Это может уменьшить

ваши затраты (вы платите только за ресурсы, которые используете) и устраняет необходимость в выделении и администрировании инфраструктуры. Тем не менее бессерверные возможности, предоставляемые поставщиками облачных вычислений, часто связаны со сроками (ограничениями времени выполнения и памяти) или недостаточно гибки (не могут управлять аппаратным обеспечением, на котором будет работать ваш код). Kubernetes предоставляет также серверные возможности после создания кластера. Доступны несколько фреймворков разной степени завершенности:

- ❑ Fast-netes;
- ❑ Nuclio.io;
- ❑ Apache OpenWhisk;
- ❑ Platform9 Fission;
- ❑ Kubless.io.

Это отличная новость для людей, работающих на Kubernetes на «голом железе» или нуждающихся в большей гибкости, чем обеспечивают облачные платформы.

Резюме

В этой главе мы рассмотрели будущее Kubernetes, и оно великолепно! Техническая основа, сообщество, широкая поддержка и импульс развития — все это очень впечатляет. Kubernetes еще молода, но темпы инноваций и устойчивости очень обнадеживают. Принципы модулирования и расширяемости Kubernetes позволяют ей стать универсальной основой для современных облачных приложений.

На данном этапе у вас должно сложиться четкое представление о том, каковы сейчас позиции Kubernetes и чем они подкреплены. Kubernetes является не просто перспективной технологией — это лидирующая платформа для оркестрации контейнеров, и такую роль она застолбила за собой на многие годы вперед. В будущем вы увидите интеграцию со все большим количеством решений и сред.

Теперь вам решать, как использовать то, что вы узнали, и создавать потрясающие вещи с Kubernetes!

Джиджи Сайфан

**Осваиваем Kubernetes.
Оркестрация контейнерных архитектур**

Перевел с английского *С. Черников*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>Е. Павлович, Е. Рафалюк-Бузовская</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.12.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 32,250. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Погрoбная информация здеcь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, гoб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гoб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гoб. 6217;
e-mail: kuznetsov@piter.com





КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы, (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами.

Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com

Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com