Kubernetes в действии

Марко Лукша

ДМК Пресс издательство

MANNING

Марко Лукша

Kubernetes в действии

Kubernetes in Action

MARKO LUKŠA

MANNING
SHELTER ISLAND

Kubernetes в действии

Марко Лукша

Перевод с английского Логунов А. В.

Москва, 2019

УДК 004.457
ББК 32.973.2
Л84

Л84 Марко Лукша
Kubernetes в действии / пер. с англ. А. В. Логунов. – М.: ДМК Пресс, 2019. – 672 с.: ил.

ISBN 978-5-97060-657-5

Книга детально рассказывает о Kubernetes – открытом программном обеспечении Google для автоматизации развёртывания, масштабирования и управления приложениями. Поддерживает основные технологии контейнеризации, также возможна поддержка технологий аппаратной виртуализации. Дано пошаговое разъяснение принципов работы и устройства модулей фреймворка. Вы узнаете все о создании объектов верхнего уровня, развертывании кластера на собственной рабочей машине и построении федеративного кластера в нескольких дата-центрах. Также детально проанализированы задачи обеспечения безопасности в Kubernetes.

Издание будет интересно всем, для кого актуальны проблемы организации кластеров и автоматизации развёртывания, масштабирования и управления приложениями.

УДК 004.457
ББК 32.973.2

Original English language edition published by Manning Publications. Copyright © 2018 by Manning Publications. Russian language edition copyright © 2018 by DMK Press. All rights reserved.

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

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

ISBN 978-5-97060-657-5 (англ.)
ISBN 978-5-97060-642-1 (рус.)

Copyright © 2018 by Manning Publications Co.
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2019

Моим родителям,
которые всегда ставили потребности своих детей
выше собственных

Оглавление

Вступительное слово от компании ITSumma 19
Описанные/рассмотренные в книге ресурсы Kubernetes 20
Благодарности 22
Предисловие 23
Признательности 25
Об этой книге 27
Кто должен читать эту книгу 27
Как организована эта книга: дорожная карта 27
О коде 29
Книжный форум 30
Другие интернет-ресурсы 30
Об авторе 32
Об иллюстрации на обложке 33
Глава 1. Знакомство с Kubernetes 34
1.1 Объяснение необходимости системы наподобие Kubernetes 36
1.1.1 Переход от монолитных приложений к микросервисам 36
1.1.2 Обеспечение консистентного окружения для приложений 39
1.1.3 Переход к непрерывной доставке: DevOps и NoOps 40
1.2 Знакомство с контейнерными технологиями 42
1.2.1 Что такое контейнеры 42
1.2.2 Знакомство с контейнерной платформой Docker 46
1.2.3 Знакомство с rkt – альтернативой Docker 50
1.3 Первое знакомство с Kubernetes 51
1.3.1 Истоки 51
1.3.2 Взгляд на Kubernetes с вершины горы 52
1.3.3 Архитектура кластера Kubernetes 54
1.3.4 Запуск приложения в Kubernetes 55
1.3.5 Преимущества использования Kubernetes 57
1.4 Резюме 60
Глава 2. Первые шаги с Docker и Kubernetes 61
2.1 Создание, запуск и совместное использование образа контейнера 61
2.1.1 Установка Docker и запуск контейнера Hello World 62
2.1.2 Создание тривиального приложения Node.js 64
2.1.3 Создание файла Dockerfile для образа 65

2.1.4 Создание контейнерого образа 66
2.1.5 Запуск образа контейнера 68
2.1.6 Исследование работающего контейнера изнутри 69
2.1.7 Остановка и удаление контейнера 71
2.1.8 Отправка образа в хранилище образов 72
2.2 Настройка кластера Kubernetes 73
2.2.1 Запуск локального одноузлового кластера Kubernetes с помощью minikube 74
2.2.2 Использование кластера Kubernetes, предоставляемого как сервис с Google Kubernetes Engine 76
2.2.3 Настройка псевдонима и автозавершение в командной строке для kubectl 79
2.3 Запуск первого приложения на Kubernetes 80
2.3.1 Развертывание приложения Node.js 80
2.3.2 Доступ к веб-приложению 83
2.3.3 Логические части вашей системы 85
2.3.4 Горизонтальное масштабирование приложения 87
2.3.5 Исследование узлов, на которых запущено приложение 90
2.3.6 Знакомство с панелью управления Kubernetes 91
2.4 Резюме 93

Глава 3. Модули: запуск контейнеров в Kubernetes 94
3.1 Знакомство с модулями 94
3.1.1 Зачем нужны модули 95
3.1.2 Общее представление о модулях 96
3.1.3 Правильная организация контейнеров между модулями 98
3.2 Создание модулей из дескрипторов YAML или JSON 100
3.2.1 Исследование дескриптора YAML существующего модуля 101
3.2.2 Создание простого дескриптора YAML для модуля 103
3.2.3 Использование команды kubectl create для создания модуля 105
3.2.4 Просмотр журналов приложений 106
3.2.5 Отправка запросов в модуль 107
3.3 Организация модулей с помощью меток 108
3.3.1 Знакомство с метками 108
3.3.2 Указание меток при создании модуля 110
3.3.3 Изменение меток существующих модулей 111
3.4 Перечисление подмножеств модулей посредством селекторов меток 112
3.4.1 Вывод списка модулей с помощью селектора меток 112
3.4.2 Использование нескольких условий в селекторе меток 113
3.5 Использование меток и селекторов для ограничения планирования модулей 114
3.5.1 Использование меток для классификации рабочих узлов 115
3.5.2 Приписывание модулей к определенным узлам 115

3.5.3 Планирование размещения на один конкретный узел 116
3.6 Аннотирование модулей 116
3.6.1 Поиск аннотаций объекта.. 117
3.6.2 Добавление и изменение аннотаций... 117
3.7 Использование пространств имен для группирования ресурсов........ 118
3.7.1 Необходимость пространств имен 118
3.7.2 Обнаружение других пространств имен и их модулей.................. 119
3.7.3 Создание пространства имен 120
3.7.4 Управление объектами в других пространствах имен................... 121
3.7.5 Изоляция, обеспечиваемая пространствами имен 121
3.8 Остановка и удаление модулей.. 122
3.8.1 Удаление модуля по имени 122
3.8.2 Удаление модулей с помощью селекторов меток........................... 122
3.8.3 Удаление модулей путем удаления всего пространства имен......... 123
3.8.4 Удаление всех модулей в пространстве имен при сохранении пространства имен.. 123
3.8.5 Удаление (почти) всех ресурсов в пространстве имен 124
3.9 Резюме 125

Глава 4. Контроллер репликации и другие контроллеры: развертывание управляемых модулей..126

4.1 Поддержание модулей в здоровом состоянии 127
4.1.1 Знакомство с проверками живучести... 128
4.1.2 Создание проверки живучести на основе HTTP 128
4.1.3 Просмотр проверки живучести в действии..................................... 129
4.1.4 Настройка дополнительных свойств проверки живучести............. 131
4.1.5 Создание эффективных проверок живучести.................................. 132
4.2 Знакомство с контроллерами репликации 134
4.2.1 Работа контроллера репликации 135
4.2.2 Создание контроллера репликации.. 137
4.2.3 Просмотр контроллера репликации в действии.............................. 138
4.2.4 Перемещение модулей в область и из области действия контроллера репликации... 142
4.2.5 Изменение шаблона модуля 145
4.2.6 Горизонтальное масштабирование модулей 146
4.2.7 Удаление контроллера репликации... 148
4.3 Использование набора реплик вместо контроллера репликации....... 149
4.3.1 Сравнение набора реплик с контроллером репликации 150
4.3.2 Определение набора реплик 150
4.3.3 Создание и исследование набора реплик.. 151
4.3.4 Использование более выразительных селекторов меток набора реплик 152
4.3.5 Подведение итогов относительно наборов реплик 153

4.4 Запуск ровно одного модуля на каждом узле с помощью набора демонов (DaemonSet) ... 153
4.4.1 Использование набора демонов для запуска модуля на каждом узле .. 154
4.4.2 Использование набора демонов для запуска модуля только на определенных узлах ... 155
4.5 Запуск модулей, выполняющих одну заканчиваемую задачу ... 158
4.5.1 Знакомство с ресурсом Job ... 158
4.5.2 Определение ресурса Job ... 159
4.5.3 Просмотр того, как задание управляет модулем ... 160
4.5.4 Запуск нескольких экземпляров модуля в задании ... 161
4.5.5 Ограничение времени, отпускаемого на завершение модуля задания .. 163
4.6 Планирование выполнения заданий периодически или один раз в будущем ... 163
4.6.1 Создание ресурса CronJob ... 163
4.6.2 Общие сведения о выполнении запланированных заданий ... 165
4.7 Резюме ... 165

Глава 5. Службы: обеспечение клиентам возможности обнаруживать модули и обмениваться с ними информацией ... 167
5.1 Знакомство со службами ... 168
5.1.1 Создание служб ... 169
5.1.2 Обнаружение служб ... 176
5.2 Подключение к службам, находящимся за пределами кластера ... 180
5.2.1 Знакомство с конечными точками служб ... 180
5.2.2 Настройка конечных точек службы вручную ... 181
5.2.3 Создание псевдонима для внешней службы ... 182
5.3 Предоставление внешним клиентам доступа к службам ... 183
5.3.1 Использование службы NodePort ... 184
5.3.2 Обеспечение доступа к службе через внешнюю подсистему балансировки нагрузки ... 188
5.3.3 Особенности внешних подключений ... 190
5.4 Обеспечение доступа к службам извне через ресурс Ingress ... 192
5.4.1 Создание ресурса Ingress ... 194
5.4.2 Доступ к службе через Ingress ... 195
5.4.3 Обеспечение доступа ко множеству служб через один и тот же Ingress.. 196
5.4.4 Настройка входа для обработки трафика TLS ... 197
5.5 Сигналы о готовности модуля к приему подключений ... 200
5.5.1 Знакомство с проверкой готовности ... 200
5.5.2 Добавление в модуль проверки готовности ... 202
5.5.3 Что должны делать реальные проверки готовности ... 204
5.6 Использование служб без обозначенной точки входа (Headless-сервисов) для обнаружения индивидуальных модулей ... 205
5.6.1 Создание службы без обозначенной точки входа ... 206

5.6.2 Обнаружение модулей через DNS ... 206
5.6.3 Обнаружение всех модулей – даже тех, которые не готовы ... 208
5.7 Устранение неполадок в службах ... 208
5.8 Резюме ... 209

Глава 6. Тома: подключение дискового хранилища к контейнерам ... 211
6.1 Знакомство с томами ... 212
6.1.1 Объяснение томов на примере ... 212
6.1.2 Знакомство с типами томов ... 214
6.2 Использование томов для обмена данными между контейнерами ... 215
6.2.1 Использование тома emptyDir ... 215
6.2.2 Использование репозитория Git в качестве отправной точки для тома ... 219
6.3 Доступ к файлам в файловой системе рабочего узла ... 222
6.3.1 Знакомство с томом hostPath ... 222
6.3.2 Исследование системных модулей с томами hostPath ... 223
6.4 Использование постоянного хранилища ... 224
6.4.1 Использование постоянного диска GCE Persistent Disk в томе модуля ... 225
6.4.2 Использование томов других типов с базовым постоянным хранилищем ... 228
6.5 Отделение модулей от базовой технологии хранения ... 230
6.5.1 Знакомство с томами PersistentVolume и заявками PersistentVolumeClaim ... 231
6.5.2 Создание ресурса PersistentVolume ... 232
6.5.3 Подача заявки на PersistentVolume путем создания ресурса PersistentVolumeClaim ... 233
6.5.4 Использование заявки PersistentVolumeClaim в модуле ... 236
6.5.5 Преимущества использования томов PersistentVolume и заявок ... 237
6.5.6 Повторное использование постоянных томов ... 238
6.6 Динамическое резервирование томов PersistentVolume ... 240
6.6.1 Определение доступных типов хранилища с помощью ресурсов StorageClass ... 240
6.6.2 Запрос на класс хранилища в заявке PersistentVolumeClaim ... 241
6.6.3 Динамическое резервирование без указания класса хранилища ... 243
6.7 Резюме ... 246

Глава 7. Словари конфигурации (ConfigMap) и секреты (Secret): настройка приложений ... 248
7.1 Конфигурирование контейнерных приложений ... 248
7.2 Передача в контейнеры аргументов командной строки ... 250
7.2.1 Определение команды и аргументов в Docker ... 250
7.2.2 Переопределение команды и аргументов в Kubernetes ... 252
7.3 Настройка переменных среды для контейнера ... 254

7.3.1 Указание переменных среды в определении контейнера 255
7.3.2 Ссылка на другие переменные среды в значении переменной 256
7.3.3 Отрицательная сторона жесткого кодирования переменных среды 256
7.4 Отсоединение конфигурации с помощью словаря конфигурации ConfigMap 257
7.4.1 Знакомство со словарями конфигурции 257
7.4.2 Создание словаря конфигурации 258
7.4.3 Передача записи словаря конфигурации в контейнер в качестве переменной среды 260
7.4.4 Одновременная передача всех записей словаря конфигурации как переменных среды 262
7.4.5 Передача записи словаря конфигурации в качестве аргумента командной строки 263
7.4.6 Использование тома configMap для обеспечения доступа к записям словаря конфигурации в виде файлов 264
7.4.7 Обновление конфигурации приложения без перезапуска приложения 271
7.5 Использование секретов для передачи чувствительных данных в контейнеры 274
7.5.1 Знакомство с секретами 274
7.5.2 Знакомство с секретом default-token 275
7.5.3 Создание секрета 277
7.5.4 Сравнение словарей конфигурации и секретов 277
7.5.5 Использование секрета в модуле 279
7.5.6 Секреты для выгрузки образов 284
7.6 Резюме 285

Глава 8. Доступ к метаданным модуля и другим ресурсам из приложений 287
8.1 Передача метаданных через нисходящий API 287
8.1.1 Доступные для использования метаданные 288
8.1.2 Предоставление доступа к метаданным через переменные среды 289
8.1.3 Передача метаданных через файлы в том downwardAPI 292
8.2 Обмен с сервером API Kubernetes 296
8.2.1 Исследование REST API Kubernetes 297
8.2.2 Обмен с сервером API изнутри модуля 302
8.2.3 Упрощение взаимодействия сервера API с контейнерами-посредниками 308
8.2.4 Использование клиентских библиотек для обмена с сервером API...... 310
8.3 Резюме 314

Глава 9. Развертывания: декларативное обновление приложений 315
9.1 Обновление приложений, работающих в модулях 316
9.1.1 Удаление старых модулей и замена их новыми 317
9.1.2 Запуск новых модулей, а затем удаление старых 317

9.2 Выполнение автоматического плавного обновления с помощью контроллера репликации 319
9.2.1 Запуск первоначальной версии приложения 319
9.2.2 Выполнение плавного обновления с помощью kubectl 321
9.2.3 Почему плавное обновление kubectl rolling-update устарело 326
9.3 Использование развертываний для декларативного обновления приложений 327
9.3.1 Создание развертывания 328
9.3.2 Обновление с помощью развертывания 331
9.3.3 Откат развертывания 335
9.3.4 Управление скоростью выкладки 338
9.3.5 Приостановка процесса выкладки 341
9.3.6 Блокировка раскруток плохих версий 342
9.4 Резюме 348

Глава 10. Ресурсы StatefulSet: развертывание реплицируемых приложений с внутренним состоянием 349
10.1 Репликация модулей с внутренним состоянием 349
10.1.1 Запуск множества реплик с отдельным хранилищем для каждой 350
10.1.2 Обеспечение стабильной долговременной идентификации для каждого модуля 352
10.2 Набор модулей с внутренним состоянием 353
10.2.1 Сопоставление наборов модулей с внутренним состоянием и наборов реплик 353
10.2.2 Обеспечение стабильной сетевой идентичности 355
10.2.3 Обеспечение стабильного выделенного хранилища для каждого экземпляра с внутренним состоянием 357
10.2.4 Гарантии набора StatefulSet 360
10.3 Использование набора StatefulSet 360
10.3.1 Создание приложения и образа контейнера 361
10.3.2 Развертывание приложения посредством набора StatefulSet 362
10.3.3 Исследование своих модулей 367
10.4 Обнаружение соседей в наборе StatefulSet 371
10.4.1 Реализация обнаружения соседей посредством DNS 373
10.4.2 Обновление набора StatefulSet 375
10.4.3 Опробование кластеризованного хранилища данных 376
10.5 Как наборы StatefulSet справляются с аварийными сбоями узлов ... 377
10.5.1 Симулирование отключения узла от сети 377
10.5.2 Удаление модуля вручную 379
10.6 Резюме 381

Глава 11. Внутреннее устройство Kubernetes 382
11.1 Архитектура 382

11.1.1 Распределенная природа компонентов Kubernetes 383
11.1.2 Как Kubernetes использует хранилище etcd 386
11.1.3 Что делает сервер API 390
11.1.4 Как сервер API уведомляет клиентов об изменениях ресурсов 392
11.1.5 Планировщик 393
11.1.6 Знакомство с контроллерами, работающими в менеджере контроллеров 396
11.1.7 Что делает агент Kubelet 401
11.1.8 Роль служебного сетевого прокси системы Kubernetes 403
11.1.9 Знакомство с надстройками Kubernetes 404
11.1.10 Все воедино 406
11.2 Взаимодействие контроллеров 406
11.2.1 Какие компоненты задействованы 406
11.2.2 Цепь событий 406
11.2.3 Наблюдение за событиями кластера 408
11.3 Что такое запущенный модуль 410
11.4 Интермодульное сетевое взаимодействие 411
11.4.1 Как должна выглядеть сеть 411
11.4.2 Более детальное рассмотрение работы сетевого взаимодействия 413
11.4.3 Знакомство с контейнерным сетевым интерфейсом 415
11.5 Как реализованы службы 416
11.5.1 Введение в kube-proxy 416
11.5.2 Как kube-proxy использует правила iptables 416
11.6 Запуск высокодоступных кластеров 418
11.6.1 Обеспечение высокой доступности приложений 418
11.6.2 Обеспечение высокой доступности компонентов плоскости управления Kubernetes 419
11.7 Резюме 423

Глава 12. Защита сервера API Kubernetes 424
12.1 Аутентификация 424
12.1.1 Пользователи и группы 425
12.1.2 Знакомство с учетными записями службы 426
12.1.3 Создание учетных записей ServiceAccount 427
12.1.4 Назначение модулю учетной записи службы 430
12.2 Защита кластера с помощью управления ролевым доступом 432
12.2.1 Знакомство с плагином авторизации RBAC 432
12.2.2 Знакомство с ресурсами RBAC 434
12.2.3 Использование ролей и привязок ролей 437
12.2.4 Применение кластерных ролей (ClusterRole) и привязок кластерных ролей (ClusterRoleBinding) 441
12.2.5 Кластерные роли и привязки кластерных ролей, существующие по умолчанию 452
12.2.6 Предоставление разумных авторизационных разрешений 455

12.3 Резюме ... 456

Глава 13. Защита узлов кластера и сети ... 457

13.1 Использование в модуле пространств имен хоста ... 457
13.1.1 Использование в модуле сетевого пространства имен узла ... 458
13.1.2 Привязка к порту хоста без использования сетевого пространства имен хоста ... 459
13.1.3 Использование пространств имен PID и IPC узла ... 461
13.2 Конфигурирование контекста безопасности контейнера ... 462
13.2.1 Выполнение контейнера от имени конкретного пользователя ... 464
13.2.2 Недопущение работы контейнера в качестве root ... 464
13.2.3 Выполнение модулей в привилегированном режиме ... 465
13.2.4 Добавление отдельных функциональных возможностей ядра в контейнер ... 467
13.2.5 Удаление функциональных возможностей из контейнера ... 469
13.2.6 Недопущение записи процессами в файловую систему контейнера ... 470
13.2.7 Совместное использование томов, когда контейнеры запущены под разными пользователями ... 471
13.3 Ограничение использования функциональности, связанной с безопасностью в модулях ... 474
13.3.1 Знакомство с ресурсами PodSecurityPolicy ... 474
13.3.2 Политики runAsUser, fsGroup и supplementalGroups ... 477
13.3.3 Конфигурирование разрешенных, стандартных и запрещенных возможностей ... 479
13.3.4 Ограничение типов томов, которые модули могут использовать ... 481
13.3.5 Назначение разных политик PodSecurityPolicy разным пользователям и группам ... 481
13.4 Изоляция сети модулей ... 485
13.4.1 Активация изоляции сети в пространстве имен ... 486
13.4.2 Разрешение подключения к серверному модулю только некоторых модулей в пространстве имен ... 486
13.4.3 Изоляция сети между пространствами имен Kubernetes ... 487
13.4.4 Изоляция с использованием обозначения CIDR ... 488
13.4.5 Лимитирование исходящего трафика набора модулей ... 489
13.5 Резюме ... 490

Глава 14. Управление вычислительными ресурсами модулей ... 491

14.1 Запрос на ресурсы для контейнеров модуля ... 491
14.1.1 Создание модулей с ресурсными запросами ... 492
14.1.2 Как ресурсные запросы влияют на назначение модуля узлу ... 493
14.1.3 Как запросы на ЦП влияют на совместное использование процессорного времени ... 498
14.1.4 Определение и запрос настраиваемых ресурсов ... 499
14.2 Лимитирование ресурсов, доступных контейнеру ... 500

14.2.1 Установка жесткого лимита на объем ресурсов, которые может использовать контейнер ... 500
14.2.2 Превышение лимитов ... 502
14.2.3 Как приложения в контейнерах видят лимиты ... 503
14.3 Классы QoS модулей ... 505
14.3.1 Определение класса QoS для модуля ... 505
14.3.2 Какой процесс уничтожается при нехватке памяти ... 508
14.4 Установка стандартных запросов и лимитов для модулей в расчете на пространство имен ... 510
14.4.1 Знакомство с ресурсом LimitRange ... 510
14.4.2 Создание объекта LimitRange ... 511
14.4.3 Обеспечение лимитов ... 513
14.4.4 Применение стандартных ресурсных запросов и лимитов ... 513
14.5 Лимитирование общего объема ресурсов, доступных в пространстве имен ... 514
14.5.1 Объект ResourceQuota ... 515
14.5.2 Указание квоты для постоянного хранилища ... 517
14.5.3 Лимитирование количества создаваемых объектов ... 517
14.5.4 Указание квот для конкретных состояний модулей и/или классов QoS ... 519
14.6 Мониторинг потребления ресурсов модуля ... 520
14.6.1 Сбор и извлечение фактических данных потребления ресурсов ... 520
14.6.2 Хранение и анализ исторической статистики потребления ресурсов ... 523
14.7 Резюме ... 526

Глава 15. Автоматическое масштабирование модулей и узлов кластера ... 528
15.1 Горизонтальное автомасштабирование модуля ... 529
15.1.1 Процесс автомасштабирования ... 529
15.1.2 Масштабирование на основе задействованности ЦП ... 533
15.1.3 Масштабирование на основе потребления памяти ... 540
15.1.4 Масштабирование на основе других, а также настраиваемых метрик ... 541
15.1.5 Определение метрик, подходящих для автомасштабирования ... 544
15.1.6 Уменьшение масштаба до нуля реплик ... 544
15.2 Вертикальное автомасштабирование модуля ... 545
15.2.1 Автоматическое конфигурирование ресурсных запросов ... 545
15.2.2 Модификация ресурсных запросов во время работы модуля ... 545
15.3 Горизонтальное масштабирование узлов кластера ... 546
15.3.1 Знакомство с кластерным автопреобразователем масштаба ... 546
15.3.2 Активация кластерного автопреобразователя масштаба ... 548
15.3.3 Ограничение прерывания службы во время уменьшения

масштаба кластера ... 549
15.4 Резюме ... 550

Глава 16. Продвинутое назначение модулей узлам ... 552
16.1 Использование ограничений и допусков для отделения модулей от определенных узлов ... 552
16.1.1 Знакомство с ограничениями и допусками ... 553
16.1.2 Добавление в узел индивидуально настроенных ограничений ... 555
16.1.3 Добавление в модули допусков ... 556
16.1.4 Для чего можно использовать ограничения и допуски ... 557
16.2 Использование сходства узлов для привлечения модулей к определенным узлам ... 558
16.2.1 Указание жестких правил сходства узлов ... 559
16.2.2 Приоретизация узлов при назначении модуля ... 561
16.3 Совместное размещение модулей с использованием сходства и антисходства модулей ... 565
16.3.1 Использование межмодульного сходства для развертывания модулей на одном узле ... 565
16.3.2 Развертывание модулей в одной стойке, зоне доступности или географическом регионе ... 568
16.3.3 Выражение предпочтений сходства модулей вместо жестких требований ... 569
16.3.4 Назначение модулей на удалении друг от друга с помощью антисходства модулей ... 571
16.4 Резюме ... 573

Глава 17. Рекомендации по разработке приложений ... 574
17.1 Соединение всего вместе ... 574
17.2 Жизненный цикл модуля ... 576
17.2.1 Приложения должны ожидать, что они могут быть удалены и перемещены ... 576
17.2.2 Переназначение мертвых или частично мертвых модулей ... 579
17.2.3 Запуск модулей в определенном порядке ... 581
17.2.4 Добавление обработчиков жизненного цикла ... 583
17.2.5 Выключение модуля ... 588
17.3 Обеспечение правильной обработки всех клиентских запросов ... 591
17.3.1 Предотвращение прерывания клиентских подключений при запуске модуля ... 592
17.3.2 Предотвращение прерванных подключений при выключении модуля ... 592
17.4 Упрощение запуска приложений и управления ими в Kubernetes ... 597
17.4.1 Создание управляемых образов контейнеров ... 597
17.4.2 Правильное тегирование образов и рациональное использование политики imagePullPolicy ... 598

17.4.3 Использование многомерных меток вместо одномерных 598
17.4.4 Описание каждого ресурса с помощью аннотаций 599
17.4.5 Предоставление информации о причинах прекращения процесса 599
17.4.6 Работа с журналами приложений 601
17.5 Рекомендации по разработке и тестированию 603
17.5.1 Запуск приложений за пределами Kubernetes во время разработки... 603
17.5.2 Использование Minikube в разработке 605
17.5.3 Версионирование и автоматическое развертывание ресурсных манифестов 606
17.5.4 Знакомство с Ksonnet как альтернативой написанию манифестов YAML/JSON 607
17.5.5 Использование непрерывной интеграции и непрерывной доставки (CI/CD) 608
17.6 Резюме 608

Глава 18. Расширение системы Kubernetes 610
18.1 Определение своих собственных объектов API 610
18.1.1 Знакомство с определениями CustomResourceDefinition 611
18.1.2 Автоматизация пользовательских ресурсов с помощью пользовательских контроллеров 615
18.1.3 Валидация пользовательских объектов 620
18.1.4 Предоставление пользовательского сервера API для пользовательских объектов 621
18.2 Расширение Kubernetes с помощью каталога служб Kubernetes (Kubernetes Service Catalog) 623
18.2.1 Знакомство с каталогом служб 623
18.2.2 Знакомство с сервером API каталога служб и менеджером контроллеров 625
18.2.3 Знакомство с брокерами служб и API OpenServiceBroker 625
18.2.4 Резервирование и использование службы 627
18.2.5 Отвязывание и дерезервирование 630
18.2.6 Что дает каталог служб 631
18.3 Платформы, построенные поверх Kubernetes 631
18.3.1 Контейнерная платформа Red Hat OpenShift 631
18.3.2 Deis Workflow и Helm 635
18.4 Резюме 638

Приложение A. Использование kubectl с несколькими кластерами.....639
A.1 Переключение между Minikube и Google Kubernetes Engine 639
A.2 Использование инструмента kubectl со множеством кластеров или пространств имен 640
A.2.1 Настройка расположения файла kubeconfig 640
A.2.2 Содержимое файла kubeconfig 640
A.2.3 Вывод списка, добавление и изменение записей в файле kubeconfig..... 642

A.2.4 Использование инструмента kubectl с разными кластерами, пользователями и контекстами 643
A.2.5 Переключение между контекстами 644
A.2.6 Перечисление контекстов и кластеров 644
A.2.7 Удаление контекстов и кластеров 644

Приложение B. Настройка многоузлового кластера с помощью kubeadm 645

B.1 Настройка ОС и необходимых пакетов 645
B.1.1 Создание виртуальной машины 645
B.1.2 Настройка сетевого адаптера для виртуальной машины 646
B.1.3 Инсталляция операционной системы 647
B.1.4 Установка Docker и Kubernetes 650
B.1.5 Клонирование виртуальной машины 652
B.2 Конфигурирование ведущего узла с помощью kubeadm 654
B.2.1 Как kubeadm запускает компоненты 655
B.3 Настройка рабочих узлов с помощью kubeadm 656
B.3.1 Настройка контейнерной сети 657
B.4 Использование кластера с локальной машины 658

Приложение C. Использование других контейнерных сред выполнения 659

C.1 Замена Docker на rkt 659
C.1.1 Настройка Kubernetes для использования rkt 659
C.1.2 Опробирование платформы rkt с Minikube 660
C.2 Использование других контейнерных сред выполнения посредством CRI 662
C.2.1 Знакомство с контейнерной средой выполнения CRI-O 663
C.2.2 Запуск приложений на виртуальных машинах вместо контейнеров 663

Приложение D. Кластерная федерация 664

D.1 Знакомство с кластерной федерацией Kubernetes 664
D.2 Архитектура 665
D.3 Федеративные объекты API 665
D.3.1 Знакомство с федеративными версиями ресурсов Kubernetes 665
D.3.2 Что делают федеративные ресурсы 666

Предметный указатель 669

Вступительное слово от компании ITSumma

Любимой кофейной кружке придется подвинуться – эта книга займет центральное место на вашем столе и станет главным навигатором в контейнерных технологиях. Мы очень рады, что книга выходит на русском языке. После своего триумфального появления в 2014 году Kubernetes основательно утвердился в качестве лидирующей программы для оркестровки контейнеров в мире. Но в России до сих пор не было ресурса, который дал бы исчерпывающую информацию о программе.

Данная книга дает пошаговое разъяснение принципов работы контейнеризации и устройства модулей в Kubernetes. Вы узнаете все о создании объектов верхнего уровня, развертывании кластера Kubernetes на собственной рабочей машине и построении федеративного кластера в нескольких дата-центрах.

Автор книги – инженер-программист с двадцатилетним стажем Марко Лукша. Последние пять лет он курировал новые разработки Kubernetes для компании Red Hat и изучил все тонкости создания приложений в данной среде. Сложные рабочие схемы Лукша преподносит в очень занимательной и последовательной манере. Вы узнаете о трех способах обеспечить доступ к службам извне и разберете варианты перехода к процессу обновления с нулевым временем простоя. А также сравните размещение в Kubernetes классических контейнеризированных микросервисов и систем, использующих внутреннее состояние.

Нас приятно удивило, что Лукша уделяет особое внимание извечному спору сисадминов и разработчиков и в том числе рассказывает о практике NoOps, которая разрешает конфликты двух лагерей. Отдельное спасибо автору за то, что он детально анализирует задачи обеспечения безопасности в Kubernetes – крайне актуальную и малоисследованную на сегодня задачу.

Наглядные схемы и «разбор полетов» в книге создают впечатление индивидуального мастер-класса. Автор предвосхищает большинство вопросов, заранее объясняя, в каком направлении двигаться, если что-то пойдет не так. Те, кто только начал знакомство с платформой, смогут уверенно перейти к практической работе в Kubernetes. А «бывалые» специалисты узнают причины и пути решения для ранее безвыходных ситуаций.

Эта книга – отличный компас в увлекательном изучении облачных технологий, которые с каждым годом набирают все больше последователей. Надеемся увидеть вас в их числе.

Ваша ITSumma

Описанные/рассмотренные в книге ресурсы Kubernetes

	Ресурс (сокр.) [версия API]	Описание	Раздел
	Namespace* (ns) [v1]	Позволяет организовывать ресурсы в неперекрывающиеся группы (например, для каждого потребителя ресурсов)	3.7
Развертывающие рабочие нагрузки	**Pod** (po) [v1]	Основная развертываемая единица, содержащая один или более процессов в расположенных рядом контейнерах	3.1
	ReplicaSet (rs) [apps/v1beta2**]	Поддерживает одну или несколько реплик модуля	4.3
	ReplicationController (rc) [v1]	Более старый, менее мощный эквивалент ресурса ReplicaSet	4.2
	Job [batch/v1]	Запускает модули, выполняющие завершаемую задачу	4.5
	CronJob [batch/v1beta1]	Запускает назначаемое задание один раз или периодически	4.6
	DaemonSet (ds) [apps/v1beta2**]	Запускает одну реплику модуля в расчете на узел (на всех узлах или только на тех, которые соответствуют селектору узлов)	4.4
	StatefulSet (sts) [apps/v1beta1**]	Запускает модули, имеющие внутреннее состояние, со стабильной идентичностью	10.2
	Deployment (deploy) [apps/v1beta1**]	Декларативное развертывание и обновление модулей	9.3
Службы	**Service** (svc) [v1]	Предоставляет доступ к одному или нескольким модулям на одной и стабильной паре IP-адреса и порта	5.1
	Endpoints (ep) [v1]	Определяет, к каким модулям (или другим серверам) предоставляется доступ через службу	5.2.1
	Ingress (ing) [extensions/v1beta1]	Предоставляет внешним клиентам доступ к одной или нескольким службам через один доступный извне IP-адрес	5.4
Конфигурация	**ConfigMap** (cm) [v1]	Словарь в формате «ключ-значение» для хранения незащищенных параметров конфигурации приложений и предоставления им доступа к ним	7.4
	Secret [v1]	Как и словарь конфигурации ConfigMap, но для конфиденциальных данных	7.5
Хранение	**PersistentVolume*** (pv) [v1]	Указывает на постоянное хранилище, которое можно смонтировать в модуль посредством заявки PersistentVolumeClaim	6.5
	PersistentVolumeClaim (pvc) [v1]	Запрос и заявка на PersistentVolume	6.5
	StorageClass* (sc) [storage.k8s.io/v1]	Определяет тип динамически резервируемого хранилища, заявляемого в PersistentVolumeClaim	6.6

(Окончание)

	Ресурс (сокр.) [версия API]	Описание	Раздел
Масштабирование	**HorizontalPodAutoscaler** (hpa) [autoscaling/v2beta1**]	Автоматически масштабирует количество реплик модулей на основе использования ЦП или другой метрики	15.1
	PodDisruptionBudget (pdb) [policy/v1beta1]	Определяет минимальное количество модулей, которые должны оставаться запущенными при эвакуации узлов	15.3.3
Ресурсы	**LimitRange** (limits) [v1]	Определяет мин, макс, ограничения и запросы по умолчанию (default) для модулей в пространстве имен	14.4
	ResourceQuota (quota) [v1]	Определяет объем вычислительных ресурсов, доступных для модулей в пространстве имен	14.5
Состояние кластера	**Node*** (нет)[v1]	Представляет рабочий узел Kubernetes	2.2.2
	Cluster* [federation/v1beta1]	Кластер Kubernetes (используемый в федерации кластеров)	Прил. D
	ComponentStatus* (cs) [v1]	Статус компонента контрольной панели	11.1.1
	Event (ev) [v1]	Отчет о том, что произошло в кластере	11.2.3
Безопасность	**ServiceAccount** (sa) [v1]	Учетная запись, используемая приложениями, запущенными в модулях	12.1.2
	Role [rbac.authorization.k8s.io/v1]	Определяет, какие действия субъект может выполнять с какими ресурсами (в расчете на пространство имен)	12.2.3
	ClusterRole* [rbac.authorization.k8s.io/v1]	Как Role, но для ресурсов уровня кластера или для предоставления доступа к ресурсам во всех пространствах имен	12.2.4
	RoleBinding [rbac.authorization.k8s.io/v1]	Определяет, кто может выполнять действия, определенные в Role или ClusterRole (в пространстве имен)	12.2.3
	ClusterRoleBinding* [rbac.authorization.k8s.io/v1]	Как RoleBinding, но по всем пространствам имен	12.2.4
	PodSecurityPolicy* (psp) [extensions/v1beta1]	Ресурс уровня кластера, который определяет, какие чувствительные для безопасности особенности могут использовать модули	13.3.1
	NetworkPolicy (netpol) [networking.k8s.io/v1]	Изолирует сеть между модулями, указывая, какие модули могут подключаться друг к другу	13.4
	CertificateSigningRequest* (csr) [certificates.k8s.io/v1beta1]	Запрос на подписание сертификата открытого ключа	5.4.4
Расш.	**CustomResourceDefinition*** (crd) [apiextensions.k8s.io/v1beta1]	Определяет настраиваемый ресурс, позволяющий пользователям создавать экземпляры настраиваемого ресурса	18.1

* Ресурс уровня кластера (без пространства имен).

** Также в других версиях API; указанная версия использована в этой книге.

Благодарности

Издательство «ДМК Пресс» благодарит компанию «ITSumma» и ее генерального директора Евгения Потапова за большую неоценимую помощь в подготовке этой книги.

Без них вы, возможно, не так хорошо поняли Kubernetes, но теперь мы полностью уверены, что все термины стоят на своих местах.

Предисловие

После работы в Red Hat в течение нескольких лет, в конце 2014 года я был назначен в недавно созданную команду под названием Cloud Enablement. Нашей задачей было вывести линейку продуктов компании промежуточного уровня на OpenShift Container Platform, которая затем разрабатывалась поверх платформы Kubernetes. В то время Kubernetes все еще находилась в зачаточном состоянии – версия 1.0 еще даже не была выпущена.

Наша команда должна была узнать все входы и выходы Kubernetes, чтобы быстро установить правильное направление для нашего программного обеспечения и воспользоваться всем, что система Kubernetes должна была предложить. Когда мы сталкивались с проблемой, нам было трудно сказать, делаем ли мы что-то неправильно или просто попали на одну из ранних ошибок Kubernetes.

С тех пор и Kubernetes, и мое понимание этой платформы прошли долгий путь. Когда я впервые начал ее использовать, большинство людей даже не слышало о Kubernetes. Теперь об этой платформе знает практически каждый инженер-программист, и она стала одним из самых быстрорастущих и широко распространенных способов запуска приложений как в облаке, так и в локальных центрах обработки данных.

В первый месяц работы с Kubernetes я написал двухсоставной блог-пост о том, как запускать кластер сервера приложений Jboss WildFly в OpenShift/Kubernetes. В то время я совсем не мог себе представить, что простой пост в блоге в конечном итоге приведет к тому, что представители издательства Manning свяжутся со мной по поводу того, хочу ли я написать книгу о Kubernetes. Конечно, я не мог сказать «нет» такому предложению, хотя я был уверен, что они также обратились к другим специалистам и в конечном итоге выбрали кого-то другого.

И все же вот мы здесь. После более чем полутора лет написания и исследования книга закончена. Это было потрясающее путешествие. Написание книги о технологии является идеальным способом узнать ее гораздо подробнее, чем если вы узнаете ее как простой пользователь. По мере того как во время процесса написания книги мои знания о Kubernetes расширялись, а сама платформа Kubernetes эволюционировала, я постоянно возвращался к предыдущим, уже написанным главам и добавлял дополнительную информацию. Я перфекционист, поэтому никогда не буду абсолютно доволен книгой, но я рад узнать, что многие читатели программы раннего доступа Manning (MEAP) считают ее отличным путеводителем по Kubernetes.

Моя цель состоит в том, чтобы помочь читателю понять саму технологию и научить их использовать ее инструментарий для эффективной и действенной разработки и развертывания приложений в кластерах Kubernetes. В книге я не уделяю особого внимания тому, как на самом деле создавать и поддерживать

надлежащий высокодоступный кластер Kubernetes, но последняя часть должна дать читателям очень четкое понимание того, из чего состоит такой кластер, и должна позволить им легко разбираться в дополнительных ресурсах, которые касаются этой темы.

Очень надеюсь, что вам понравится читать эту книгу и что она научит вас получать максимальную отдачу от удивительной системы под названием Kubernetes.

Признательности

До того, как я начал писать эту книгу, я понятия не имел, сколько людей будет вовлечено в ее перенос из грубой рукописи в опубликованную работу. А это означает, что я просто обязан поблагодарить очень многих людей.

Во-первых, я хотел бы поблагодарить Эрин Твохи за то, что она обратилась ко мне с просьбой написать эту книгу, и Майкла Стивенса из Мэннинга, который с первого дня был полностью уверен в моей способности написать ее. Его слова ободрения на раннем этапе действительно мотивировали меня и поддерживали меня в течение последних полутора лет.

Также хотел бы поблагодарить моего технического редактора Эндрю Уоррена, который помог мне выпустить мою первую главу, и Элешу Хайд, которая затем сменила Эндрю и работала со мной вплоть до последней главы. Спасибо за то, что вы со мной, хотя со мной трудно иметь дело, поскольку я, как правило, довольно регулярно выпадаю с радаров.

Кроме того, хотел бы поблагодарить Жанну Боярскую, ставшую первым рецензентом, которая читала и комментировала мои главы, пока я их писал. Жанна и Элеша сыграли важную роль в том, чтобы эта книга, как мы надеемся, стала недурной. Без их комментариев книга никогда бы не получила таких хороших отзывов от внешних рецензентов и читателей.

Хотел бы поблагодарить моего технического корректора, Антонио Мадгаги, и, конечно, всех моих внешних рецензентов: Эла Кринкера, Алессандро Кампэйса, Александра Мыльцевы, Чабу Сари, Дэвида Димариа, Элиаса Ранхеля, Эриса Зеленка, Фабрицио Куччи, Джареда Данкана, Кэйт Дональдсон, Майкла Брайта, Паоло Антинори, Питера Перлепеса и Тиклу Гангули. Их положительные комментарии меня поддерживали время от времени, когда я беспокоился, что мой стиль был совершенно ужасным и бесполезным. С другой стороны, их конструктивная критика помогла улучшить разделы, которые я быстро собрал без достаточных усилий. Спасибо, что указали на трудные для понимания разделы и предложили способы улучшения книги. Кроме того, спасибо за правильные вопросы, которые заставили меня понять свои ошибки в двух или трех вещах в первоначальных версиях рукописи.

Мне также нужно поблагодарить читателей, которые купили раннюю версию книги через программу MEAP издательства Мэннинга и озвучили свои комментарии на онлайн-форуме или обратились ко мне напрямую, особенно Вимала Кансала, Паоло Патьерно и Роланда Хаба, которые заметили немало несоответствий и других ошибок. И я хотел бы поблагодарить всех в издательстве Manning, кто участвовал в издании этой книги. Прежде чем я закончу, мне также нужно поблагодарить моего коллегу и друга школьных лет Эла Джастина, который привел меня в Red Hat, и моих замечательных коллег из команды Cloud Enablement. Если бы я не был в Red Hat или в этой команде, я бы не написал эту книгу.

Наконец, я хотел бы поблагодарить мою жену и моего сына, которые относились ко мне с пониманием и поддерживали меня в течение последних 18 месяцев, когда я запирался в своем офисе, вместо того чтобы проводить время с ними.

Спасибо всем!

Об этой книге

Основная задача книги «*Kubernetes в действии*» состоит в том, чтобы сделать вас опытным пользователем Kubernetes. Он научит вас практически всем понятиям, в которых вам нужно разбираться, чтобы эффективно разрабатывать и запускать приложения в среде Kubernetes.

Перед погружением в Kubernetes данная книга дает обзор контейнерных технологий, таких как Docker, в том числе того, как строить контейнеры, так что даже читатели, которые не использовали эти технологии раньше, могут с нуля организовать и вести свою работу. Затем она неспешно проведет вас через значительную часть того, что вам нужно знать о Kubernetes, – от элементарных понятий до вещей, скрытых под поверхностью.

Кто должен читать эту книгу

Данная книга посвящена в первую очередь разработчикам приложений, но также содержит обзор управляющих приложений с точки зрения системного администрирования. Она предназначена для тех, кто заинтересован в запуске и управлении контейнеризированными приложениями на более чем одном сервере.

Как начинающие, так и продвинутые инженеры-программисты, которые хотят познакомиться с контейнерными технологиями и организовать несколько связанных контейнеров в масштабе, получат опыт, необходимый для разработки, контейнеризации и запуска своих приложений в среде Kubernetes.

Предыдущий опыт работы с контейнерными технологиями либо Kubernetes вовсе не обязателен. Книга объясняет данный предмет, постепенно углубляясь в подробности, и не использует никакого исходного кода приложений, который было бы слишком трудно понять разработчикам, не являющимся экспертами.

Вместе с тем читатели должны иметь, по крайней мере, базовые знания в области программирования, компьютерных сетей и выполнения основных команд в Linux, а также понимание общеизвестных компьютерных протоколов, таких как HTTP.

Как организована эта книга: дорожная карта

Эта книга состоит из трех частей, которые включают в себя 18 глав.

Часть 1 дает краткое введение в Docker и Kubernetes, как настроить кластер Kubernetes и как запустить в нем простое приложение. Она содержит две главы:

- глава 1 объясняет, что такое Kubernetes, как эта система появилась и как она помогает решать сегодняшние задачи управления приложениями в масштабе;

- глава 2 представляет собой практическое руководство по созданию образа контейнера и по его запуску в кластере Kubernetes. В ней также объясняется, как запускать локальный одноузловой кластер Kubernetes и надлежащий многоузловой кластер в облаке.

В части 2 представлены ключевые понятия, в которых необходимо разбираться для запуска приложений в Kubernetes. Разделы этой части следующие:

- глава 3 вводит фундаментальный строительный блок в Kubernetes – модуль (Pod) и объясняет, как организовать модули и другие объекты Kubernetes посредством меток (Label);
- глава 4 знакомит вас с тем, как Kubernetes поддерживает приложения в рабочем состоянии путем автоматического перезапуска контейнеров. В ней также показывается, как правильно запускать управляемые модули, горизонтально масштабировать их, делать их устойчивыми к сбоям узлов кластера и запускать их в заданное время в будущем или периодически;
- в главе 5 показано, как модули могут обеспечивать доступ к службе (Service), которую они предоставляют клиентам, работающим как внутри, так и за пределами кластера. В ней также показано, как модули, работающие в кластере, могут обнаруживать службы и получать к ним доступ независимо от того, находятся они в кластере или нет;
- в главе 6 объясняется, как несколько контейнеров, работающих в одном модуле, могут совместно использовать файлы и как управлять постоянным хранилищем и делать его доступным для модулей;
- в главе 7 показано, как передавать конфигурационные данные и конфиденциальную информацию, такую как учетные данные, приложениям, работающим внутри модулей;
- глава 8 описывает, как приложения могут получать информацию о среде Kubernetes, в которой они работают, и как они могут общаться с Kubernetes для изменения состояния кластера;
- глава 9 знакомит с концепцией развертывания (Deployment) и объясняет, как правильно запускать и обновлять приложения в среде Kubernetes;
- в главе 10 представлен специальный способ запуска приложений с сохранением состояния, которые обычно требуют конкретной идентификации и хранения данных о собственном состоянии.

Часть 3 глубоко погружается во внутренние части кластера Kubernetes, вводит некоторые дополнительные понятия и рассматривает все, что вы узнали в первых двух частях, с более высокой точки зрения. Это последняя группа глав:

- глава 11 опускается под поверхность Kubernetes и объясняет все компоненты, которые составляют кластер Kubernetes, и что каждый из них делает. В ней также объясняется, как модули обмениваются по сети и как службы выполняют балансировку нагрузки между несколькими модулями;

- в главе 12 объясняется, как защитить API Kubernetes и в более общем смысле кластер с помощью аутентификации и авторизации;
- в главе 13 рассказывается, как модули могут получать доступ к ресурсам узла и как администратор кластера может это предотвратить;
- глава 14 посвящена ограничению вычислительных ресурсов, потребляемых каждым приложением, настройке гарантий качества обслуживания приложений и мониторингу использования ресурсов отдельными приложениями. Она также знакомит вас с тем, как препятствовать тому, чтобы пользователи потребляли слишком много ресурсов;
- в главе 15 рассказывается, как настроить Kubernetes на автоматическое масштабирование числа запущенных реплик приложения, а также как увеличивать размер кластера, если текущее число узлов кластера не может принимать дополнительные приложения;
- в главе 16 показано, как обеспечить назначение модулей только определенным узлам или как воспрепятствовать их назначению другим узлам. Она также показывает, как убедиться, что модули назначены вместе, или как не допускать этого;
- глава 17 знакомит вас с тем, как нужно разрабатывать свои приложения, чтобы сделать их добропорядочными гражданами вашего кластера. В ней также дается несколько советов о том, как настроить рабочие процессы разработки и тестирования для уменьшения накладных расходов;
- в главе 18 показано, как можно расширять Kubernetes с помощью собственных настраиваемых объектов и как это сделали другие и создали платформы приложений корпоративного уровня.

По мере прохождения этих глав вы не только узнаете об отдельных строительных блоках Kubernetes, но и постепенно улучшите свои знания об использовании инструмента командной строки `kubectl`.

О коде

Хотя эта книга не содержит массу фактического исходного кода, в ней действительно много манифестов ресурсов Kubernetes в формате YAML и команд оболочки вместе с их результатами. Все это отформатировано `вот таким моноширинным шрифтом`, чтобы отделить его от обычного текста.

Команды оболочки в основном выделены **жирным шрифтом**, чтобы четко отделить их от результатов их выполнения, но иногда жирным шрифтом для акцента выделены только наиболее важные части команды или части результатов команды. В большинстве случаев результаты команды были переформатированы, чтобы уместить их в ограниченное пространство книги. Кроме того, поскольку инструмент командного интерфейса `kubectl` системы Kubernetes постоянно развивается, новые версии могут печатать больше информации, чем показано в книге. Не смущайтесь, если они не совпадают в точности.

Листинги иногда включают маркер продолжения строки (➥), чтобы показать, что строка текста переносится на следующую строку. Они также включают аннотации, которые выделяют и объясняют наиболее важные части.

В текстовых абзацах некоторые очень распространенные элементы, такие как Pod, ReplicationController, ReplicaSet, DaemonSet и т. д., задаются обычным шрифтом, чтобы избежать чрезмерного распространения шрифта кода и способствовать читаемости. В некоторых местах слово «Pod» пишется в верхнем регистре для ссылки на ресурс Pod, а в нижнем регистре для ссылки на фактический модуль, то есть группу запущенных контейнеров.

Все образцы в книге были протестированы с помощью Kubernetes версии 1.8, работающей в Google Kubernetes Engine и в локальном кластере под управлением Minikube. Полный исходный код и YAML манифестов можно найти на `https://github.com/luksa/kubernetes-in-action` либо скачать с веб-сайта издателя в `www.manning.com/books/kubernetes-in-action`.

Книжный форум

Покупка книги «*Kubernetes в действии*» включает в себя бесплатный доступ к частному веб-форуму под управлением Manning Publications, где вы можете комментировать книгу, задавать технические вопросы и получать помощь от автора и других пользователей. Для того чтобы получить доступ к форуму, перейдите по ссылке `https://forums.manning.com/forums/kubernetes-in-action`. Вы также можете узнать больше о форумах издательства Manning и правилах поведения на `https://forums.manning.com/forums/about`.

Обязательство издательства Manning перед нашими читателями заключается в том, чтобы обеспечить место, где может иметь место содержательный диалог между отдельными читателями и между читателями и автором. Это не является обязательством какого-либо конкретного участия со стороны автора, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам попробовать задать автору несколько сложных вопросов, чтобы его интерес не пропал! Форум и архив предыдущих обсуждений будут доступны с веб-сайта издателя до тех пор, пока книга находится в печати.

Другие интернет-ресурсы

Вы можете найти широкий спектр дополнительных ресурсов Kubernetes в следующих местах:

- сайт Kubernetes на `https://kubernetes.io`;
- блог Kubernetes, в котором регулярно публикуется интересная информация (`http://blog.kuber-netes.io`);
- канал Slack сообщества Kubernetes на `http://slack.k8s.io`;
- каналы YouTube системы Kubernetes и фонда Cloud Native Computing Foundation:

- https://www.youtube.com/channel/UCZ2bu0qutTOM0tHYa_jkIwg;
- https://www.youtube.com/channel/UCvqbFHwN-nwalWPjPUKpvTA.

Для того чтобы получить более глубокое понимание отдельных тем или даже помочь внести свой вклад в Kubernetes, вы также можете обратиться в любую из специальных групп по интересам (SIGs), связанных с Kubernetes на https://github.com/kubernetes/kubernetes/wiki/Special-Interest-Groups-(SIGs).

И наконец, поскольку Kubernetes является системой с открытым исходным кодом, есть множество информации, доступной в самом исходном коде Kubernetes. Вы найдете его на https://github.com/kubernetes/kubernetes и в родственных репозиториях.

Об авторе

Марко Лукша – инженер-программист с более чем 20-летним профессиональным опытом разработки всего, от простых веб-приложений до полных ERP-систем, вычислительных платформ и программного обеспечения среднего уровня. Он сделал свои первые шаги в программировании еще в 1985 году, в возрасте шести лет, на подержанном компьютере ZX Spectrum, который ему купил отец. В начальной школе он был национальным чемпионом в конкурсе программирования на языке Logo и посещал летние лагеря для программистов, где он научился программировать на Pascal. С тех пор он занимался разработкой программного обеспечения на широком спектре языков программирования.

В старших классах он начал создавать динамические веб-сайты, когда интернет был еще относительно молод. Затем он перешел к разработке программного обеспечения для отрасли здравоохранения и телекоммуникаций в местной компании, изучая информатику в университете Любляны, Словения. В конце концов, он перешел на работу в Red Hat, первоначально разрабатывая реализацию с открытым исходным кодом API Google App Engine, в которой использовались продукты Red Hat среднего уровня JBoss. Он также работал или участвовал в таких проектах, как CDI/Weld, Infinispan/Jboss DataGrid и др.

С конца 2014 года он является частью команды Red Hat Cloud Enablement, где его обязанности включают в себя обновление новых разработок в Kubernetes и связанных с ними технологий и обеспечение максимального задействования возможностей Kubernetes и OpenShift в программном обеспечении компании среднего уровня.

Об иллюстрации на обложке

Фигура на обложке «*Kubernetes в действии*» является «членом дивана», турецкого государственного совета или руководящего органа. Иллюстрация взята из коллекции костюмов Османской империи, опубликованной 1 января 1802 года Уильямом Миллером с Олд Бонд-стрит в Лондоне. Титульный лист отсутствует в коллекции, и мы не смогли отследить его до настоящего времени. Оглавление книги идентифицирует цифры на английском и французском языках, и каждая иллюстрация носит имена двух художников, которые над ней работали, оба из которых, несомненно, были бы удивлены, обнаружив, что их рисунок украшает обложку книги компьютерного программирования… 200 лет спустя.

Коллекция была приобретена редактором Manning на антикварном блошином рынке в «Гараже» на 26-й Западной улице Манхэттена. Продавец был американцем, проживающем в Анкаре, Турция, и сделка состоялась как раз в тот момент, когда он собирал свой стенд в конце рабочего дня. У редактора издательства Manning не было при себе существенной суммы наличных, необходимой для покупки, а кредитная карта и чек были вежливо отклонены. Ситуация становилась безнадежной. Каково было решение проблемы? Это оказалось не более чем старомодным устным соглашением, скрепленным рукопожатием. Продавец предложил перевести ему деньги безналично, и редактор вышел с рынка с банковской информацией на бумажке и портфолио изображений под мышкой. Излишне говорить, что мы перечислили средства на следующий день, и мы по-прежнему благодарны и впечатлены доверием этого неизвестного человека.

Мы в Manning приветствуем изобретательность, инициативу и, да, удовольствие от компьютерного бизнеса с книжными обложками, основанными на богатом разнообразии региональной жизни два столетия назад, возвращенными к жизни картинами из этой коллекции.

Глава 1

Знакомство с Kubernetes

Эта глава посвящена:

- описанию изменений в разработке и развертывании программного обеспечения за последние годы;
- изолированию приложений и уменьшению средовых различий с помощью контейнеров;
- описанию того, как контейнеры и Docker используются в Kubernetes;
- упрощению работы разработчиков и сисадминов с помощью Kubernetes.

Много лет назад многие программные приложения были большими монолитами, работающими либо как один процесс, либо как небольшое количество процессов, распределенных по нескольким серверам. Сегодня такие унаследованные системы по-прежнему широко распространены. Они имеют медленные циклы релизов и обновляются относительно редко. В конце каждого цикла релиза разработчики упаковывают всю систему и передают ее группе системных администраторов, которая затем ее развертывает и контролирует. В случае аварийных сбоев оборудования системные администраторы вручную переносят ее на оставшиеся работоспособные серверы.

Сегодня эти огромные монолитные устаревшие приложения медленно распадаются на более мелкие, самостоятельно работающие компоненты, называемые микросервисами. Поскольку микросервисы друг от друга отделены, их можно разрабатывать, развертывать, обновлять и масштабировать по отдельности. Это позволяет быстро изменять компоненты и так часто, как это необходимо, чтобы идти в ногу с сегодняшними быстро меняющимися бизнес-требованиями.

Но, учитывая большое количество развертываемых компонентов и все более крупные центры обработки данных, настраивать, управлять и поддер-

живать бесперебойную работу всей системы становится все труднее. Гораздо труднее понять, куда поместить каждый из этих компонентов, чтобы достичь высокой степени использования ресурсов и тем самым снизить затраты на оборудование. Делать все это вручную крайне тяжело. Нам нужна автоматизация, которая включает в себя автоматическое размещение этих компонентов на наших серверах, автоматическую настройку, контроль и обработку аварийных сбоев. Именно здесь в игру вступает Kubernetes.

Kubernetes позволяет разработчикам развертывать свои приложения самостоятельно и так часто, как они хотят, не требуя какой-либо помощи от системных администраторов. Но Kubernetes приносит пользу не только разработчикам. Эта платформа также помогает системным администраторам, автоматически отслеживая и перемещая эти приложения в случае аварийного сбоя оборудования. Основное внимание системных администраторов переключается с надзора за отдельными приложениями на надзор и контроль за платформой Kubernetes и остальной инфраструктурой, тогда как сама платформа Kubernetes заботится о приложениях.

ПРИМЕЧАНИЕ. *Kubernetes* на греческом языке означает «кормчий» или «рулевой» (лицо, держащее рулевое колесо корабля). Люди произносят слово Kubernetes по-разному. Многие произносят его как кубернетес, в то время как другие произносят больше как кубернетис. Независимо от того, какую форму вы используете, специалисты поймут, что вы имеете в виду.

Kubernetes абстрагирует аппаратную инфраструктуру и обеспечивает доступ ко всему вашему центру обработки данных (дата-центру) как одному огромному вычислительному ресурсу. Это позволяет развертывать и запускать программные компоненты без необходимости знать о фактических серверах, находящихся в основании. При развертывании многокомпонентного приложения с помощью Kubernetes он выбирает сервер для каждого компонента, развертывает его и позволяет легко находить и взаимодействовать со всеми другими компонентами приложения.

Все это делает Kubernetes отличным вариантом для большинства локальных дата-центров, но во всей красе эта система проявляет себя тогда, когда она используется в крупных центрах обработки данных, таких, которые построены и эксплуатируются облачными провайдерами. Kubernetes позволяет им предлагать разработчикам простую платформу для развертывания и запуска любого типа приложений, не требуя от облачного провайдера, чтобы собственные системные администраторы что-то знали о десятках тысяч приложений, работающих на их оборудовании.

Благодаря тому что все больше крупных компаний принимают модель Kubernetes как лучший способ запуска приложений, она становится стандартным способом запуска распределенных приложений как в облаке, так и в локальной инфраструктуре.

1.1 Объяснение необходимости системы наподобие Kubernetes

Прежде чем вы начнете подробно знакомиться с Kubernetes, давайте рассмотрим, как изменились разработка и развертывание приложений за последние годы. Это изменение является следствием как разделения больших монолитных приложений на более мелкие микросервисы, так и изменений в инфраструктуре, которая запускает эти приложения. Понимание данных изменений поможет вам лучше понять преимущества использования Kubernetes и контейнерных технологий, таких как Docker.

1.1.1 Переход от монолитных приложений к микросервисам

Монолитные приложения состоят из компонентов, которые тесно связаны друг с другом и должны разрабатываться, развертываться и управляться как одна сущность, поскольку все они выполняются как один процесс ОС. Изменения в одной части приложения требуют новой выкладки всего приложения, и со временем отсутствие жестких границ между частями приводит к увеличению сложности и последующему ухудшению качества всей системы из-за неограниченного роста взаимосвязей между этими частями.

Для запуска монолитного приложения обычно требуется небольшое количество мощных серверов, которые могут предоставить достаточно ресурсов для запуска приложения. Для того чтобы справиться с растущей нагрузкой на систему, вам нужно либо масштабировать серверы вертикально (так называемое масштабирование вверх), добавляя больше процессоров, оперативной памяти и других серверных компонентов, либо масштабировать всю систему по горизонтали, настраивая дополнительные серверы и запуская несколько копий (или реплик) приложения (масштабирование вширь). Хотя масштабирование вверх обычно не требует каких-либо изменений в приложении, оно относительно быстро становится дорогостоящим и на практике всегда имеет верхний предел. Масштабирование вширь, с другой стороны, является относительно дешевым аппаратно, но может потребовать больших изменений в программном коде приложения и не всегда возможно – некоторые части приложения с большим трудом поддаются горизонтальному масштабированию или почти невозможны для него (например, реляционные базы данных). Если какая-либо часть монолитного приложения не масштабируется, то все приложение становится немасштабируемым, если только каким-то образом этот монолит не разделить.

Разделение приложений на микросервисы

Эти и другие проблемы заставили нас начать разбиение сложных монолитных приложений на небольшие независимые развертывания компонентов, называемых микросервисами. Каждый микросервис выполняется как неза-

висимый процесс (см. рис. 1.1) и взаимодействует с другими микросервисами через простые, четко определенные интерфейсы (API).

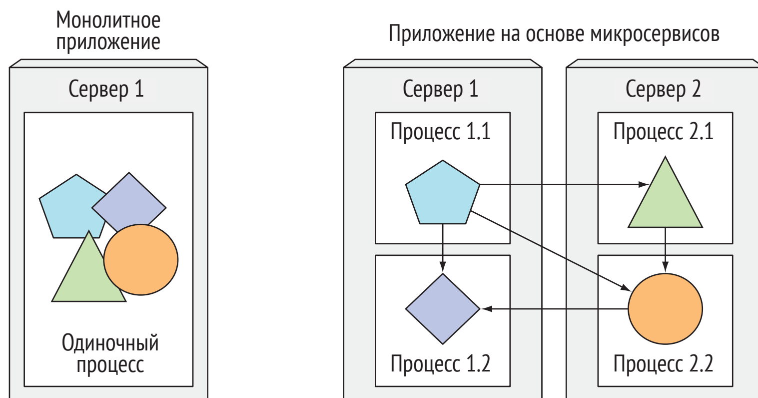


Рис. 1.1. Компоненты внутри монолитного приложения и независимых микросервисов

Микросервисы взаимодействуют через синхронные протоколы, такие как HTTP, используя которые, они обычно предоставляют RESTful, или через асинхронные протоколы, такие как AMQP (Advanced Message Queueing Protocol, расширенный протокол организации очереди сообщений). Эти протоколы просты, хорошо понятны большинству разработчиков и не привязаны к какому-либо конкретному языку программирования. Каждый микросервис может быть написан на языке, который наиболее целесообразен для реализации конкретных микросервисов.

Поскольку каждый микросервис представляет собой автономный процесс с относительно статическим внешним API, существует возможность разрабатывать и развертывать каждый микросервис отдельно. Изменение одной из них не требует изменений или повторного развертывания какого-либо другого сервиса, при условии что API не изменяется или изменяется только обратно совместимым образом.

Масштабирование микросервисов

Масштабирование микросервисов, в отличие от монолитных систем, где необходимо масштабировать систему целиком, выполняется отдельно для каждого сервиса. Это означает, что вы можете масштабировать только те сервисы, которые требуют больше ресурсов, оставляя другие сервисы в исходном масштабе. На рис. 1.2 показан пример. Некоторые компоненты реплицируются и выполняются как несколько процессов, развернутых на разных серверах, в то время как другие выполняются как один процесс приложения. Когда монолитное приложение не может быть промасштабировано вширь по причине того, что одна из ее частей немасштабируема, разделение приложения на микросервисы позволяет горизонтально масштабировать те составные части, которые позволяют масштабировать вширь, и вертикально масштабировать те части, которые нельзя масштабировать горизонтально.

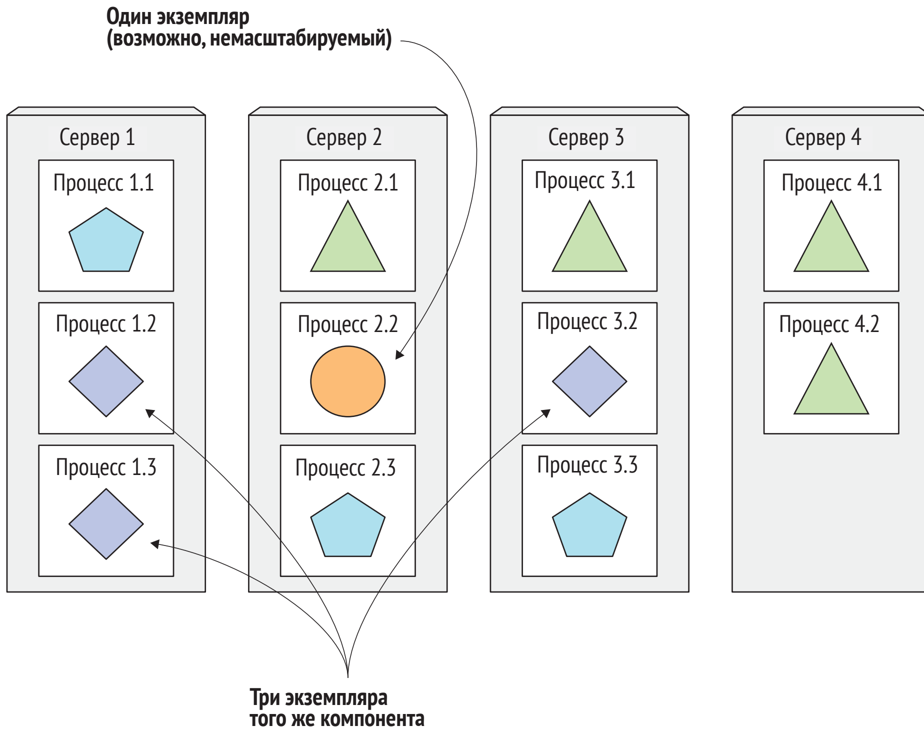


Рис. 1.2. Каждый микросервис может масштабироваться по отдельности

Развертывание микросервисов

Как это обычно бывает, микросервисы имеют свои недостатки. Если система состоит лишь из небольшого количества развертываемых компонентов, управлять этими компонентами очень просто. Вопрос, где развертывать каждый компонент, решается тривиальным образом, потому что вариантов не так много. Когда количество этих компонентов увеличивается, решения, связанные с развертыванием, становятся все более трудными, поскольку не только количество комбинаций развертывания увеличивается, но и количество взаимосвязей между компонентами увеличивается еще больше.

Микросервисы выполняют свою работу вместе, как команда, поэтому им нужно найти друг друга и пообщаться. При их развертывании кто-то или что-то должно правильно настроить их все, чтобы они могли работать вместе как одна система. С увеличением количества микросервисов это становится утомительным и подверженным ошибкам, в особенности если учесть то, что сисадмины должны делать, когда сервер аварийно завершает свою работу.

Микросервисы также создают другие проблемы, к примеру затрудняют отладку и трассировку вызовов выполнения, поскольку они охватывают несколько процессов и машин. К счастью, эти проблемы теперь решаются с помощью систем распределенного сбора трассировок, таких как Zipkin.

Понимание различий в требованиях к окружению

Как я уже упоминал, компоненты в архитектуре микросервисов не только развертываются независимо, но и разрабатываются таким же образом. Из-за их независимости и того факта, что, как правило, каждый компонент разрабатывается отдельными командами, ничто не мешает каждой команде использовать разные библиотеки и заменять их всякий раз, когда возникает необходимость. Расхождение в библиотеках между компонентами приложения, как показано на рис. 1.3, где в любом случае требуются разные версии одной библиотеки, – это неизбежность.

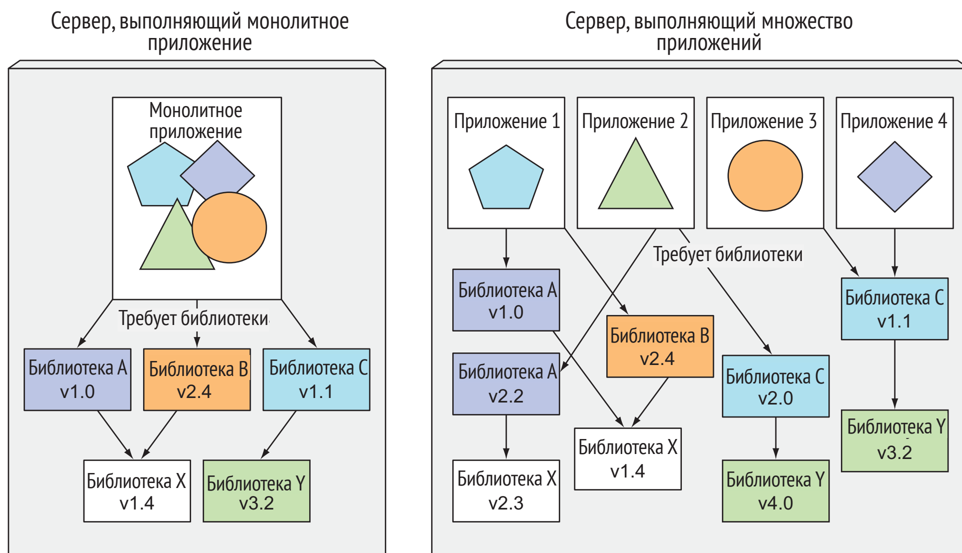


Рис. 1.3. Несколько приложений, работающих на одном хосте, могут иметь конфликтующие версии библиотек, от которых они функционально зависимы

Развертывание динамически связанных приложений, для которых требуются различные версии совместно используемых библиотек и/или другие особенности среды, может быстро стать кошмаром для сисадминов, которые развертывают их и управляют ими на рабочих серверах. Чем больше количество компонентов необходимо развернуть на одном хосте, тем сложнее будет управлять всеми их зависимостями, чтобы удовлетворить все их требования.

1.1.2 Обеспечение консистентного окружения для приложений

Независимо от того, сколько отдельных компонентов вы разрабатываете и развертываете, одна из самых больших проблем, с которой всегда приходится сталкиваться разработчикам и системным администраторам, – это различия в окружениях, в которых они выполняют свои приложения. Мало того, что существует огромная разница между окружением разработки и рабочим окру-

жением, различия даже существуют между отдельными машинами в рабочем окружении. Еще одним неизбежным фактом является то, что окружение одной рабочей машины будет меняться с течением времени.

Эти различия варьируются от аппаратного обеспечения до операционной системы и библиотек, доступных на каждой машине. Рабочие окружения управляются системными администраторами, в то время как разработчики занимаются своими ноутбуками, на которых ведется разработка, самостоятельно. Разница в том, насколько эти две группы людей знают о системном администрировании, и это, по понятным причинам, приводит к относительно большим различиям между этими двумя системами, не говоря уже о том, что системные администраторы уделяют гораздо больше внимания последним обновлениям в системе безопасности, в то время как многих разработчиков это не интересует в такой же мере.

Кроме того, рабочие системы могут выполнять приложения от нескольких разработчиков или групп разработчиков, что не обязательно верно для компьютеров разработчиков. Рабочая система должна обеспечивать надлежащую среду для всех размещаемых на ней приложений, даже если для них могут потребоваться разные, порой конфликтующие версии библиотек.

Для того чтобы уменьшить количество проблем, которые проявляются только в рабочем окружении, было бы идеально, если бы приложения могли работать в одной и той же среде во время разработки и в «бою», чтобы они имели одну и ту же операционную систему, библиотеки, конфигурацию системы, сетевую среду и все остальное. Вы также не хотите, чтобы эта среда слишком сильно менялась с течением времени, если вообще менялась. Кроме того, если это возможно, требуется способность добавлять приложения на тот же сервер, не затрагивая ни одно из существующих приложений на этом сервере.

1.1.3 Переход к непрерывной доставке: DevOps и NoOps

В последние несколько лет мы также наблюдаем изменения во всем процессе разработки приложений и в том, как приложения обслуживаются в рабочем окружении. В прошлом работа команды разработчиков состояла в создании приложения и ее передаче группе системных администраторов, которая затем его развертывала, сопровождала и поддерживала его работу. Однако теперь организации понимают, что лучше, чтобы та же команда, которая разрабатывает приложение, также принимала участие в его развертывании и сопровождала его на протяжении всей его жизни. Это означает, что команды разработчиков, QA и системные администраторы теперь должны сотрудничать на протяжении всего процесса. Подобная практика называется *DevOps*.

Преимущества

Когда разработчики принимают более активное участие в выполнении приложения в производстве, это приводит к тому, что они имеют лучшее понимание потребностей и нужд пользователей и проблем, с которыми сталкиваются

системные администраторы при сопровождении приложения. Разработчики приложений теперь также гораздо более склонны передавать пользователям приложение раньше, а затем использовать их отзывы, чтобы определять пути дальнейшего развития приложения.

Для того чтобы чаще выпускать новые версии приложений, необходимо оптимизировать процесс развертывания. В идеале требуется, чтобы разработчики развертывали приложения сами, не дожидаясь системных администраторов. Однако развертывание приложения часто требует понимания базовой инфраструктуры и организации оборудования в дата-центре. Разработчики не всегда знают эти детали и в большинстве случаев даже не хотят знать о них.

Позвольте разработчикам и сисадминам делать то, что они делают лучше всего

Несмотря на то что разработчики и системные администраторы работают для достижения одной и той же цели – запуск успешного программного приложения в качестве службы для своих клиентов, они имеют различные индивидуальные цели и мотивирующие факторы. Разработчики любят создавать новые функциональные средства и улучшать пользовательский опыт. Обычно они не хотят следить за тем, чтобы своевременно вносились исправления в систему безопасности базовой операционной системы, и т. п. Они предпочитают оставлять это на усмотрение системных администраторов.

Системные администраторы отвечают за процесс развертывания программного обеспечения в рабочем окружении и аппаратную инфраструктуру, на которой они работают. Они занимаются обеспечением безопасности системы, задействованием ресурсов и другими аспектами, которые не являются приоритетными для разработчиков. Системные администраторы не хотят иметь дело с неявными взаимозависимостями всех компонентов приложения и не хотят думать о том, как изменения операционной системы или инфраструктуры могут повлиять на работу приложения в целом, но они влияют.

В идеале требуется, чтобы разработчики развертывали приложения сами, ничего не зная об аппаратной инфраструктуре и не имея дела с системными администраторами. Это называется *NoOps*. Очевидно, что вам все же нужно, чтобы кто-то занимался аппаратной инфраструктурой, при этом, в идеале, не занимаясь особенностями каждого приложения, работающего на нем.

Как вы увидите, Kubernetes позволяет нам достичь всего этого. Абстрагируясь от фактического оборудования и обеспечивая к нему доступ как к единой платформе для развертывания и выполнения приложений, эта инфраструктура позволяет разработчикам настраивать и развертывать свои приложения без какой-либо помощи от системных администраторов и дает возможность системным администраторам сосредоточиваться на поддержании базовой инфраструктуры в рабочем состоянии, не зная ничего о реальных приложениях, работающих поверх него.

1.2 Знакомство с контейнерными технологиями

В разделе 1.1 я представил неполный список проблем, с которыми сталкиваются сегодняшние команды разработчиков и системные администраторы. Хотя имеется много способов борьбы с ними, эта книга будет сосредоточена на том, как они решаются с помощью Kubernetes.

Kubernetes использует контейнерные технологии Linux, для того чтобы обеспечить изоляцию выполняющихся приложений, поэтому, прежде чем мы углубимся в систему Kubernetes как таковую, вы должны быть знакомы с основами контейнеров, чтобы понимать, что Kubernetes делает сама, а что она сбрасывает на контейнерные технологии, такие как *Docker* или *rkt* (произносится как «рокит», от англ. «rock-it»).

1.2.1 Что такое контейнеры

В разделе 1.1.1 мы увидели, как различные программные компоненты, выполняющиеся на одной машине, будут требовать разные, возможно конфликтующие, версии библиотек, от которых они зависят, или иметь другие разные требования к среде в целом.

Когда приложение состоит лишь из небольшого количества крупных компонентов, вполне допустимо предоставить каждому компоненту выделенную виртуальную машину (VM) и изолировать их среды, предоставив каждому из них собственный экземпляр операционной системы. Но когда эти компоненты начинают уменьшаться в объеме и их количество начинает расти, становится невозможным предоставлять каждому из них свою собственную виртуальную машину, если вы не хотите тратить аппаратные ресурсы и снизить затраты на оборудование. Но дело не только в растрате аппаратных ресурсов. Поскольку каждая виртуальная машина обычно должна настраиваться и управляться индивидуально, увеличение количества виртуальных машин также приводит к трате человеческих ресурсов, поскольку они значительно увеличивают рабочую нагрузку на системных администраторов.

Изолирование компонентов с помощью контейнерных технологий Linux

Вместо того чтобы использовать виртуальные машины для изоляции сред каждого микросервиса (или программных процессов в целом), разработчики обращаются к контейнерным технологиям Linux. Данные технологии позволяют запускать несколько сервисов на одной хост-машине, не только обеспечивая доступ к разным средам, но и изолируя их друг от друга, подобно виртуальным машинам, но с гораздо меньшими затратами.

Процесс, запущенный в контейнере, выполняется внутри операционной системы хоста, как и все другие процессы (в отличие от виртуальных машин, где процессы выполняются в отдельных операционных системах). Но процесс в контейнере по-прежнему изолирован от других процессов. Для самого процесса это выглядит, как если бы он был единственным работающим на машине и в ее операционной системе.

Сравнение виртуальных машин с контейнерами

По сравнению с виртуальными машинами, контейнеры гораздо облегченнее, что позволяет запускать большее количество программных компонентов на одном и том же оборудовании, главным образом потому, что каждая виртуальная машина должна запускать свой собственный набор системных процессов, который требует еще вычислительных ресурсов в дополнение к тем, которые потребляются собственным процессом компонента. С другой стороны, контейнер – это не что иное, как отдельный изолированный процесс, выполняющийся в центральной ОС, потребляющий только те ресурсы, которые приложение потребляет, без накладных расходов в виде дополнительных процессов.

Из-за накладных расходов виртуальных машин вам в конечном счете приходится группировать несколько приложений в каждую отдельную виртуальную машину, поскольку у вас недостаточно ресурсов для выделения всей виртуальной машины каждому приложению. При использовании контейнеров, как показано на рис. 1.4, вы можете (и должны) иметь по одному контейнеру для каждого приложения. Конечным результатом является то, что вы можете на одном и том же аппаратном обеспечении умещать еще больше приложений.

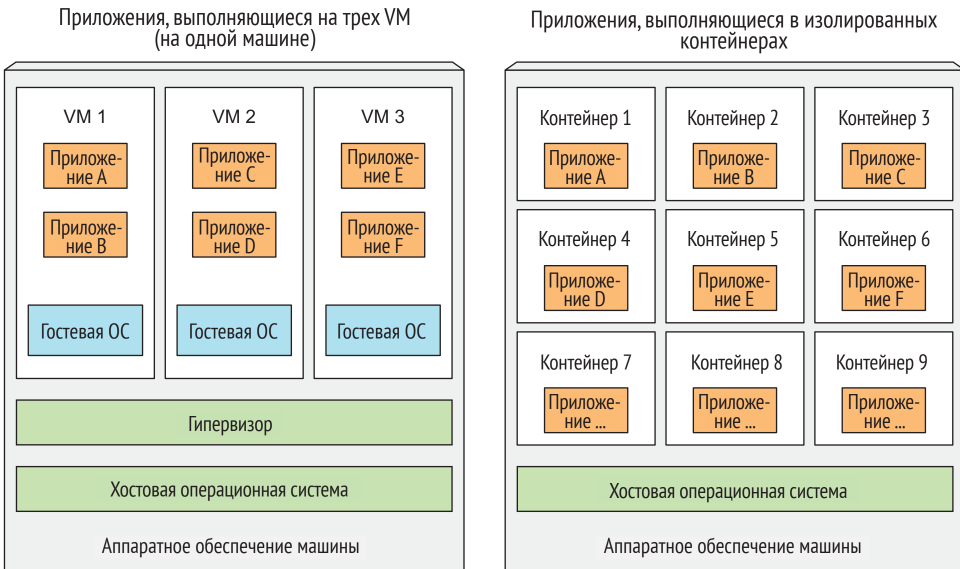


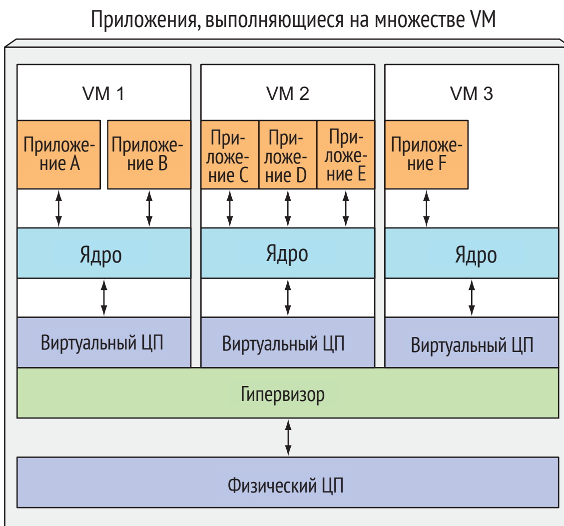
Рис. 1.4. Использование виртуальных машин для изоляции групп приложений по сравнению с изоляцией отдельных приложений с помощью контейнеров

Когда вы запускаете три виртуальные машины на хосте, у вас имеется три совершенно отделенные друг от друга операционные системы, работающие на одном и том же аппаратном обеспечении. Под этими виртуальными машинами находятся хостовая ОС и гипервизор, который разделяет физические аппаратные ресурсы на меньшие наборы виртуальных ресурсов, которые могут использоваться операционной системой внутри каждой виртуальной машины. Приложения, запущенные на этих виртуальных машинах, выполняют

системные вызовы ядра гостевой ОС в виртуальной машине, а затем ядро выполняет инструкции x86 на физическом ЦП хоста через гипервизор.

> **ПРИМЕЧАНИЕ.** Существует два типа гипервизоров, то есть программ управления операционными системами. Гипервизоры 1-го типа не используют хостовую ОС, в то время как гипервизоры 2-го типа ее используют.

Контейнеры же выполняют системные вызовы на одном и том же ядре, работающем в хостовой ОС. Это единственное ядро, выполняющее инструкции x86 на процессоре хоста. ЦП не нужно делать какой-либо виртуализации, как он делает с виртуальными машинами (см. рис. 1.5).



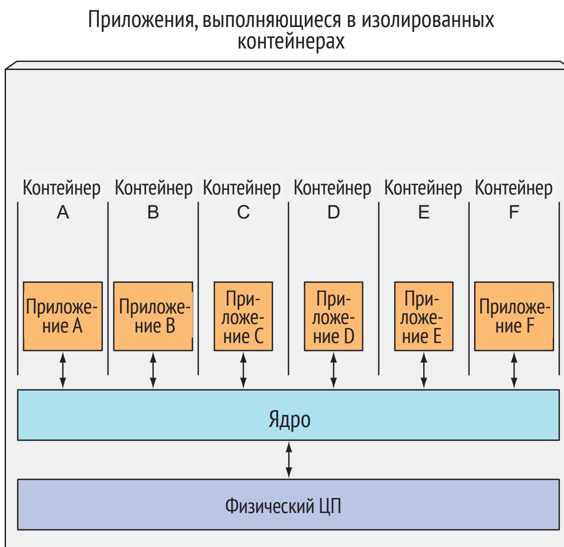


Рис. 1.5. Разница между тем, как приложения в виртуальных машинах используют процессор и как они используют их в контейнерах

Главное преимущество виртуальных машин заключается в их полной изоляции, поскольку каждая виртуальная машина работает под управлением собственного ядра Linux, в то время как все контейнеры обращаются к одному ядру, что может явно представлять угрозу безопасности. При ограниченном количестве аппаратных ресурсов виртуальные машины могут использоваться только в том случае, если требуется изолировать небольшое количество процессов. Для выполнения большего количества изолированных процессов на одной машине контейнеры являются гораздо лучшим выбором из-за их низкой нагрузки. Помните, что каждая виртуальная машина выполняет свой собственный набор системных служб, в то время как контейнеры этого не делают, потому что все они выполняются в одной ОС. Это также означает, что для запуска контейнера начальная загрузка компонентов не требуется, как в случае с виртуальными машинами. Процесс, выполняющийся в контейнере, запускается немедленно.

Знакомство с механизмами, обеспечивающими изоляцию контейнеров

К этому моменту вы, вероятно, задаетесь вопросом, как именно контейнеры могут изолировать процессы, если они работают в одной операционной системе. Это возможно благодаря двум механизмам. Первый, *пространства имен Linux*, гарантирует, что каждый процесс видит свое персональное представление о системе (файлы, процессы, сетевые интерфейсы, сетевое имя (hostname) и т. д.). Второй – *контрольные группы Linux* (*cgroups*), ограничивающие объем ресурсов, которые может потреблять процесс (ЦП, оперативная память, пропускная способность сети и т. д.).

Изоляция процессов с помощью пространств имен Linux

По умолчанию каждая система Linux изначально имеет одно пространство имен. Все системные ресурсы, такие как файловые системы, идентификаторы процессов, идентификаторы пользователей, сетевые интерфейсы и др., принадлежат одному пространству имен. Но вы можете создавать дополнительные пространства имен и организовывать ресурсы между ними. При запуске процесса вы запускаете его внутри одного из этих пространств имен. Процесс будет видеть только ресурсы, находящиеся внутри одного пространства имен. Существует несколько типов пространств имен, поэтому процесс принадлежит не одному пространству имен, а одному пространству имен каждого типа.

Существуют следующие виды пространств имен:

- файловая система (mnt);
- идентификатор процессов (pid);
- сеть (net);
- межпроцессное взаимодействие (ipc);
- UTS;
- пользовательские идентификаторы (user).

Каждый вид пространства имен используется для изоляции определенной группы ресурсов. Например, пространство имен UTS определяет, какой хостнейм и доменное имя видит процесс, запущенный внутри этого пространства имен. Назначив двум процессам два разных пространства имен UTS, можно заставить их видеть разные локальные хостнеймы. Другими словами, для двух процессов это будет выглядеть так, как будто они выполняются на двух разных машинах (по крайней мере, в том, что касается хостнеймов).

Точно так же то, к какому сетевому пространству имен принадлежит процесс, определяет, какие сетевые интерфейсы видит приложение, запущенное внутри процесса. Каждый сетевой интерфейс принадлежит только одному пространству имен, но он может быть перемещен из одного пространства имен в другое. Каждый контейнер использует свое собственное сетевое пространство имен, и поэтому каждый контейнер видит свой собственный набор сетевых интерфейсов.

Это должно дать вам общее представление о том, как пространства имен используются для изоляции приложений, выполняющихся в контейнерах, друг от друга.

Ограничение ресурсов, доступных процессу

Другая половина изоляции контейнеров связана с ограничением объема системных ресурсов, которые может потреблять контейнер. Это достигается с помощью cgroups, функционального средства ядра Linux, которое ограничивает использование ресурсов процессом (или группой процессов). Процесс не может использовать больше, чем настроенный объем ЦП, оперативной памяти, пропускной способности сети и т. д. Благодаря этому процессы не могут перехватывать ресурсы, зарезервированные для других процессов, что аналогично тому, когда каждый процесс выполняется на отдельной машине.

1.2.2 Знакомство с контейнерной платформой Docker

Хотя контейнерные технологии существуют уже давно, они стали более широко известны с появлением контейнерной платформы Docker. Docker была первой контейнерной системой, которая сделала контейнеры легко переносимыми на разные машины. Это упростило процесс упаковки не только приложения, но и всех его библиотек и других зависимостей, даже всей файловой системы ОС, в простой, переносимый пакет, который может использоваться для подготовки приложения к работе на любой другой машине, на которой работает Docker.

При выполнении приложения, упакованного с помощью Docker, оно видит точное содержимое файловой системы, поставляемое вместе с ним. Оно видит одни и те же файлы, независимо от того, работает ли оно на вашей машине, предназначенной для разработки, или же на машине из рабочего окружения, даже если на рабочем сервере запущена совершенно другая ОС Linux. Приложение не будет видеть ничего с сервера, на котором оно выполняется, поэтому не имеет значения, имеет или нет сервер совершенно другой набор установленных библиотек по сравнению с вашей машиной для разработки.

Например, если вы упаковали свое приложение с файлами всей операционной системы Red Hat Enterprise Linux (RHEL), приложение будет считать, что оно работает внутри RHEL, как при запуске на компьютере разработчика, на котором работает Fedora, так и при запуске на сервере под управлением Debian или другого дистрибутива Linux. Только ядро может отличаться.

Это похоже на создание образа виртуальной машины путем установки операционной системы на виртуальную машину, установки приложения внутри нее, а затем распространения всего образа виртуальной машины и его выполнения. Платформа Docker достигает того же эффекта, но, вместо того чтобы использовать виртуальные машины для изоляции приложений, она использует контейнерные технологии Linux, упомянутые в предыдущем разделе, чтобы обеспечить (почти) тот же уровень изоляции, что и у виртуальных машин. Вместо использования больших монолитных образов виртуальных машин применяются образы контейнеров, которые обычно меньше.

Ключевая разница между образами контейнеров на основе Docker и образами виртуальных машин заключается в том, что образы контейнеров состоят из слоев, которые можно совместно повторно использовать в нескольких разных образах. Это означает, что должны загружаться только определенные слои образа, если другие слои уже были загружены ранее при запуске другого образа контейнера, который тоже содержит те же слои.

Понятия платформы Docker

Docker – это платформа для упаковки, распространения и выполнения приложений. Как мы уже говорили, она позволяет упаковывать приложение вместе со всей его средой. Это могут быть либо несколько библиотек, которые требуются приложению, либо даже все файлы, которые обычно доступны в файловой системе установленной операционной системы. Docker позволяет переносить этот пакет в центральный репозиторий, из которого он затем может быть перенесен на любой компьютер, на котором работает Docker, и выполнен там (по большей части, но не всегда, как мы скоро объясним).

Этот сценарий состоит из трех главных понятий в Docker:

- *образы (Images)*. Образ контейнера на основе Docker – это то, во что вы упаковываете свое приложение и его среду. Он содержит файловую систему, которая будет доступна приложению, и другие метаданные, такие как путь к исполняемому файлу, который должен быть исполнен при запуске образа;
- *хранилища (Registry)*. Хранилище Docker – это репозиторий, в котором хранятся образы Docker и который упрощает обмен этими образами между различными людьми и компьютерами. Когда вы создаете образ, вы можете либо запустить его на компьютере, на котором вы его создали, либо *отправить* (закачать) образ в хранилище, а затем *извлечь* (скачать) его на другом компьютере и запустить его там. Некоторые хранилища являются общедоступными, позволяя любому извлекать из них образы, в то время как другие являются приватными, доступными только для определенных людей или машин;

- *контейнеры*. Контейнер на основе Docker – это обычный контейнер Linux, созданный из образа контейнера на основе Docker. Выполняемый контейнер – это процесс, запущенный на хосте, на котором работает Docker, но он полностью изолирован как от хоста, так и от всех других процессов, запущенных на нем. Процесс также ограничен ресурсами, имея в виду, что он может получать доступ и использовать только тот объем ресурсов (ЦП, ОЗУ и т. д.), который ему выделен.

Создание, распространение и запуск образа Docker

На рис. 1.6 показаны все три понятия и их взаимосвязь. Разработчик сначала создает образ, а затем помещает его в хранилище. Таким образом, образ доступен всем, кто имеет доступ к хранилищу. Затем он может закачать образ на любую другую машину, на которой выполняется Docker, и запустить образ. Docker создает изолированный контейнер на основе образа и выполняет двоичный исполняемый файл, указанный в рамках образа.

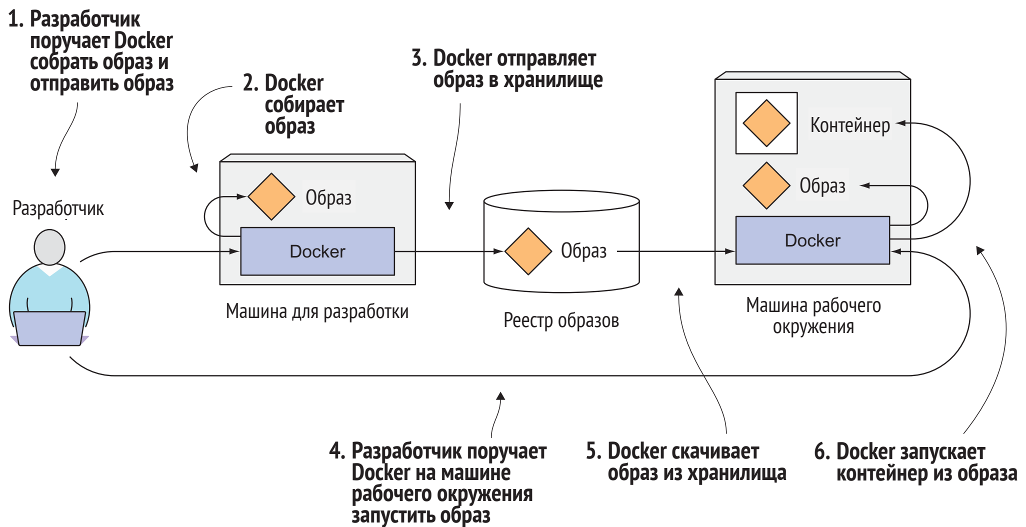
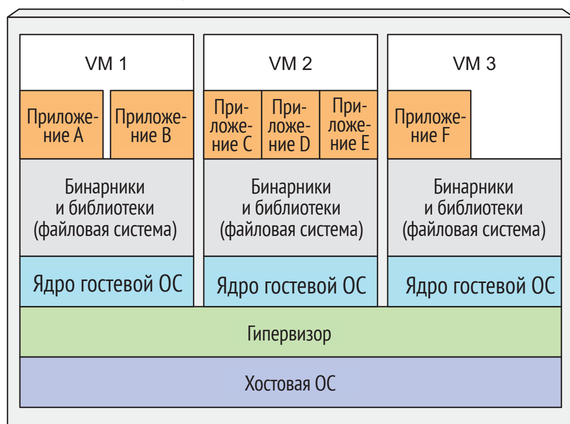


Рис. 1.6. Образы, реестры и контейнеры Docker

Сравнение виртуальных машин и контейнеров docker

Я объяснил, что контейнеры Linux в общем похожи на виртуальные машины, но гораздо облегченнее. Теперь давайте рассмотрим, чем конкретно контейнеры Docker отличаются от виртуальных машин (и чем образы Docker отличаются от образов виртуальных машин). На рис. 1.7 снова показаны те же шесть приложений, работающих как в виртуальных машинах, так и в контейнерах Docker.



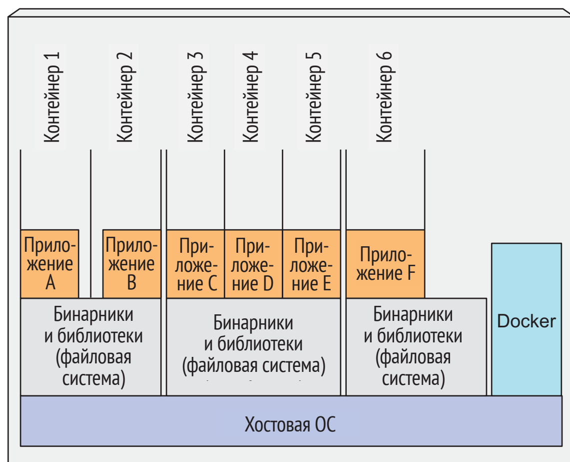


Рис. 1.7. Выполнение шести приложений на трех виртуальных машинах по сравнению с их выполнением в контейнерах Docker

Вы заметите, что приложения A и B имеют доступ к одним и тем же двоичным файлам и библиотекам как при работе в виртуальной машине, так и при работе в виде двух отдельных контейнеров. В виртуальной машине это очевидно, поскольку оба приложения видят одну и ту же файловую систему (виртуальную машину). Но мы сказали, что каждый контейнер имеет свои собственные, изолированные системы. Каким образом приложение A и приложение B могут совместно использовать одни и те же файлы?

Слои образа

Я уже говорил, что образы Docker состоят из слоев. Разные образы могут содержать одни и те же слои, поскольку каждый слой Docker надстроен поверх другого образа, а два разных образа могут использовать один и тот же

родительский образ в качестве основы. Это ускоряет распределение образов по сети, так как слои, которые уже были перенесены как часть первого образа, не нужно передавать снова при передаче другого образа.

Но слои не только делают распределение более эффективным, они также помогают уменьшить объем хранения образов. Каждый слой хранится только один раз. Таким образом, два контейнера, созданные из двух образов, основанных на одних и тех же базовых слоях, могут читать одни и те же файлы, но если один из них записывает эти файлы, другой не видит этих изменений. Таким образом, даже если они совместно используют файлы, они по-прежнему изолированы друг от друга. Это работает, поскольку слои образов контейнеров доступны только для чтения. При запуске контейнера поверх слоев образа создается новый слой, доступный для записи. Когда процесс в контейнере записывает в файл, расположенный в одном из низлежащих слоев, копия всего файла создается в верхнем слое, и процесс записывает в копию.

Ограничения переносимости образов контейнеров

Теоретически образ контейнера может быть выполнен на любой машине Linux, в которой работает Docker, но существует одно небольшое предостережение, связанное с тем, что все контейнеры, работающие на хосте, используют Linux'овское ядро хоста. Если контейнеризированное приложение требует конкретной версии ядра, оно может работать не на каждой машине. Если на машине работает другая версия ядра Linux или на ней нет доступных модулей ядра, приложение не сможет на нем работать.

Контейнеры – гораздо облегченнее, по сравнению с виртуальными машинами, вместе с тем они накладывают определенные ограничения на приложения, работающие внутри них. Виртуальные машины не имеют таких ограничений, так как каждая виртуальная машина выполняет свое собственное ядро.

И дело не только в ядре. Также должно быть ясно, что контейнеризированное приложение, созданное для определенной аппаратной архитектуры, может выполняться на других машинах только с такой же архитектурой. Вы не можете контейнеризировать приложение, созданное для архитектуры x86, и ожидать, что оно будет работать на машине на базе ARM, поскольку оно также запускает Docker. Для этого по-прежнему будет нужна виртуальная машина.

1.2.3 Знакомство с rkt – альтернативой Docker

Docker был первой контейнерной платформой, которая сделала контейнеры мейнстримом. Надеюсь, я ясно дал понять, что сама платформа Docker не обеспечивает изоляцию процесса. Фактическая изоляция контейнеров выполняется на уровне ядра Linux с помощью таких функций ядра, как пространства имен Linux и контрольные группы. Docker только упрощает использование этих функциональных возможностей.

После успеха Docker родилась инициатива Open Container Initiative (OCI) по созданию открытых отраслевых стандартов вокруг форматов и сред выполнения контейнеров. Docker является частью этой инициативы, как и rkt (произносится как «рокит», от англ. «rock-it»), который является еще одним контейнерным движком Linux.

Как и Docker, rkt является платформой для запуска контейнеров. Она уделяет особое внимание обеспечению безопасности, компонуемости и соответствию открытым стандартам. Она использует формат контейнерного образа OCI и может даже выполнять обычные образы контейнеров Docker.

Эта книга посвящена использованию платформы Docker в качестве среды запуска контейнеров для системы Kubernetes, поскольку изначально она была единственной, поддерживаемой в Kubernetes. В последнее время система Kubernetes начала поддерживать rkt, а также другие, в качестве среды запуска контейнеров.

Причина, почему я упоминаю rkt в этом месте, состоит в том, чтобы вы не сделали ошибку, подумав о том, что Kubernetes является системой оркестровки контейнеров, специально спроектированной для контейнеров на базе Docker. На самом деле в ходе этой книги вы поймете, что суть Kubernetes – не оркестровка контейнеров. Ее роль намного больше. Контейнеры являются наилучшим способом запуска приложений на разных кластерных узлах. Имея это в виду, давайте, наконец, погрузимся в суть того, чему эта книга посвящена, – Kubernetes.

1.3 Первое знакомство с Kubernetes

Мы уже показали, что по мере роста количества разворачиваемых компонентов приложений в системе становится все труднее управлять ими всеми. Google, вероятно, была первой компанией, которая поняла, что ей нужен гораздо более оптимальный способ развертывания и управления их программными компонентами и их инфраструктурой, с тем чтобы обеспечить масштабируемость в глобальном масштабе. Это одна из немногих компаний в мире, которая управляет сотнями тысяч серверов и имеет дело с управлением развертываниями в таком массовом масштабе. Что вынудило их разрабатывать решения, позволяющие сделать разработку и развертывание тысяч программных компонентов управляемым и экономичным.

1.3.1 Истоки

На протяжении многих лет разработанная в Google внутренняя система под названием Borg (а позже новая система под названием Omega) помогала как разработчикам приложений, так и системным администраторам управлять этими тысячами приложений и служб. В дополнение к упрощению процесса разработки и управления она также помогала им достигать гораздо более вы-

сокой степени задействованности их инфраструктуры, что важно, когда ваша организация настолько велика. Когда вы управляете сотнями тысяч машин, даже незначительные улучшения в задействованности означают экономию в миллионах долларов, поэтому стимулы для разработки такой системы очевидны.

Продержав целое десятилетие системы Borg и Omega в секрете, в 2014 году Google анонсировала Kubernetes, систему с открытым исходным кодом, основанную на опыте, приобретенном благодаря Borg, Omega и другим внутренним системам Google.

1.3.2 Взгляд на Kubernetes с вершины горы

Kubernetes – это программная система, которая позволяет легко развертывать контейнеризированные приложения и управлять ими. Она использует возможности контейнеров Linux для запуска разнородных приложений без необходимости знать какие-либо внутренние детали этих приложений и без необходимости вручную развертывать эти приложения на каждом хосте. Поскольку данные приложения работают в контейнерах, они не влияют на другие приложения, работающие на том же сервере, что имеет решающее значение при запуске приложений для совершенно разных организаций на одном и том же оборудовании. Это имеет первостепенное значение для облачных провайдеров, поскольку они стремятся к максимально возможной задействованности своего оборудования, сохраняя при этом полную изоляцию размещенных приложений.

Kubernetes позволяет выполнять ваши программные приложения на тысячах компьютерных узлов, как если бы все эти узлы были одним огромным компьютером. Она абстрагируется от базовой инфраструктуры и тем самым упрощает разработку, развертывание и управление как для разработчиков, так и для системных администраторов.

Процедура развертывания приложений через Kubernetes всегда одинаковая, независимо от того, содержит ли кластер всего несколько узлов или тысячи. Размер кластера не имеет никакого значения. Дополнительные узлы кластера просто представляют собой дополнительный объем ресурсов, доступных для развернутых приложений.

Суть того, что делает Kubernetes

Рисунок 1.8 показывает простейший вид системы Kubernetes. Система состоит из ведущего узла (мастера) и любого количества рабочих узлов. Когда разработчик отправляет список приложений ведущему узлу, Kubernetes развертывает их в кластере рабочих узлов. То, на какой узел приземляется компонент, не имеет (и не должно иметь) значения ни для разработчика, ни для системного администратора.

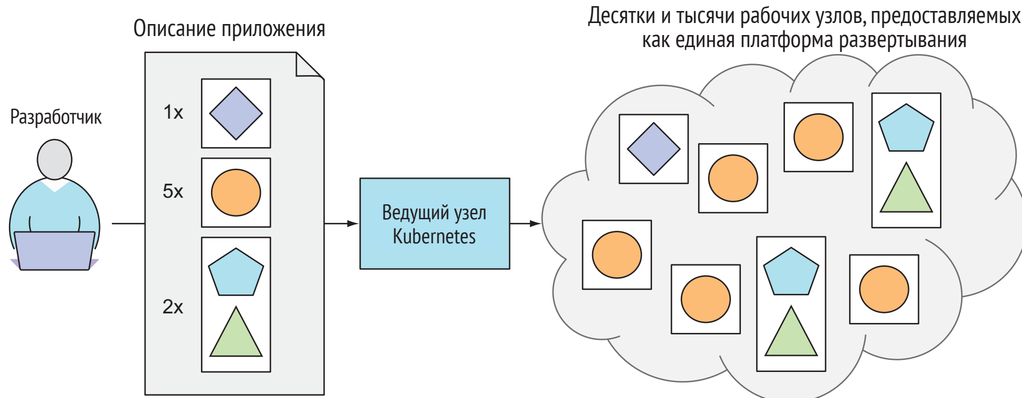


Рис. 1.8. Kubernetes обеспечивает доступ ко всему центру обработки данных как единой платформе развертывания

Разработчик может указать, что определенные приложения должны выполняться вместе, и Kubernetes развернет их на одном рабочем узле. Другие будут разбросаны по всему кластеру, но они могут обмениваться между собой одинаково, независимо от того, где они развернуты.

Помощь разработчикам в концентрации на ключевом функционале приложения

Kubernetes можно рассматривать как операционную систему для кластера. Она избавляет разработчиков приложений от необходимости внедрять в свои приложения определенные службы, связанные с инфраструктурой; вместо этого в вопросе предоставления этих служб они опираются на Kubernetes. Это включает в себя такие аспекты, как обнаружение службы, масштабирование, балансировка нагрузки, самовосстановление и даже выбор лидера. Поэтому разработчики приложений могут сосредоточиться на реализации реального функционала приложений и не тратить время на то, чтобы разбираться в том, как интегрировать их с инфраструктурой.

Помощь системным администраторам в достижении более оптимальной задействованности ресурсов

Kubernetes будет выполнять ваше контейнеризованное приложение где-нибудь в кластере, предоставит информацию его компонентам о том, как отыскать друг друга, и будет поддерживать их в работающем состоянии. Поскольку приложению все равно, на каком узле оно работает, Kubernetes может перемещать приложение в любое время и, путем смешивания и отождествления приложений, добиваться гораздо лучшей задействованности ресурсов, чем это возможно при ручном планировании.

1.3.3 Архитектура кластера Kubernetes

Мы увидели архитектуру Kubernetes с высоты птичьего полета. Теперь давайте более подробно рассмотрим, из чего состоит кластер Kubernetes. На аппаратном уровне кластер Kubernetes состоит из множества узлов, которые можно разделить на два типа:

- *ведущий* узел (мастер), на котором размещена *плоскость управления (Control Plane) Kubernetes*, контролирующая и управляющая всей системой Kubernetes;
- рабочие *узлы*, на которых выполняются развертываемые приложения.

На рис. 1.9 показаны компоненты, работающие на этих двух наборах узлов. Далее я дам по ним объяснение.

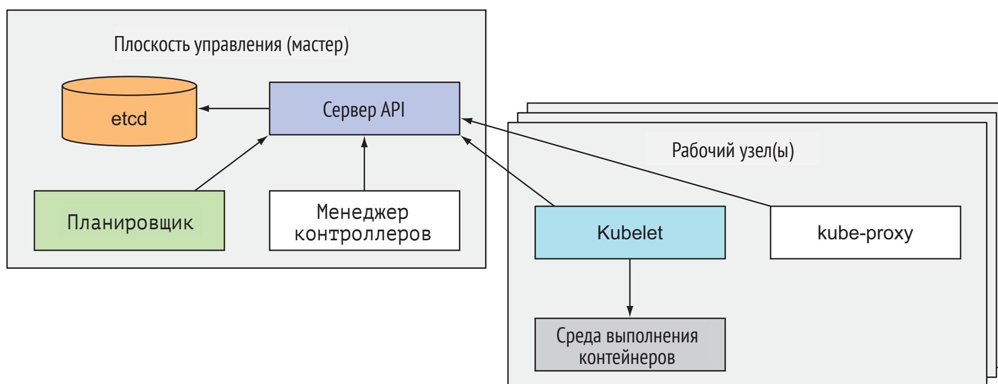


Рис. 1.9. Компоненты, составляющие кластер Kubernetes

Плоскость управления

Плоскость управления – это то, что управляет кластером и заставляет его функционировать. Она состоит из нескольких компонентов, которые могут работать на одном ведущем узле либо быть распределены по нескольким узлам и реплицированы для обеспечения высокой доступности. Эти компоненты следующие:

- *сервер Kubernetes API*, с которым взаимодействуете вы и другие компоненты плоскости управления;
- *планировщик*, который распределяет приложения (назначает рабочий узел каждому развертываемому компоненту приложения);
- *менеджер контроллеров*, выполняющий функции кластерного уровня, такие как репликация компонентов, отслеживание рабочих узлов, обработка аварийных сбоев узлов и т. д.;
- *etcd*, надежное распределенное хранилище данных, которое непрерывно сохраняет конфигурацию кластера.

Компоненты плоскости управления содержат и управляют состоянием кластера, но не выполняют приложения. Это делается (рабочими) узлами.

Узлы

Рабочие узлы – это машины, на которых выполняются контейнеризированные приложения. Задача выполнения, мониторинга и предоставления служб приложениям выполняется следующими компонентами:

- Docker, rkt или другая *среда выполнения контейнеров*, в которой выполняются контейнеры;
- *Kubelet*, агент, который обменивается с сервером API и управляет контейнерами на своем узле;
- *служебный прокси Kubernetes* (*kube-proxy*), который балансирует нагрузку сетевого трафика между компонентами приложения.

Мы подробно объясним все эти компоненты в главе 11. Я не поклонник объяснять, как все работает, прежде чем сначала объяснить, *что* тот или иной компонент делает, и научить людей его использовать. Это как научиться водить машину. Вы не хотите знать, что под капотом. Сначала вы хотите узнать, как провести ее из точки А в точку Б. Только после того, как вы узнаете, как это сделать, вы заинтересуетесь тем, как автомобиль делает это возможным. В конце концов, знание того, что находится под капотом, может когда-нибудь помочь вам снова заставить автомобиль двигаться, после того как он сломается и оставит вас на обочине дороги.

1.3.4 Запуск приложения в Kubernetes

Для того чтобы запустить приложение в Kubernetes, сначала необходимо упаковать его в один или несколько образов контейнеров, отправить эти образы в хранилище образов, а затем опубликовать описание приложения на сервере API Kubernetes.

Описание содержит такие сведения, как образ или образы контейнеров, содержащие компоненты приложения, как эти компоненты связаны друг с другом и какие из них должны выполняться совместно (вместе на одном узле), а какие нет. Для каждого компонента можно также указать, сколько копий (или *реплик*) требуется выполнять. Кроме того, это описание также включает в себя то, какой из этих компонентов предоставляет службу внутренним либо внешним клиентам и должен быть открыт через единый IP-адрес и доступен для обнаружения другим компонентам.

Как описание приводит к запуску контейнера

Когда сервер API обрабатывает описание приложения, планировщик назначает указанные группы контейнеров доступным рабочим узлам, исходя из вычислительных ресурсов, требуемых каждой группой, и нераспределенных ресурсов на каждом узле в данный момент. Агент Kubelet на этих узлах затем поручает среде выполнения контейнеров (например, Docker) извлечь из хранилища требуемые образы контейнеров и запустить контейнеры.

Изучите рис. 1.10, чтобы лучше понять, как приложения развертываются в Kubernetes. В описании приложения перечислены четыре контейнера, сгруп-

пированных в три набора (эти наборы называются *модулями* (подами), мы объясним, что это такое, в главе 3). Первые два модуля содержат всего один контейнер, тогда как последний содержит два. Это означает, что оба контейнера должны работать совместно и не должны быть изолированы друг от друга. Рядом с каждым модулем также вы увидите число, представляющее количество реплик каждого модуля, которые должны выполняться параллельно. После отправки дескриптора в Kubernetes он запланирует использование указанного количества реплик каждого модуля на доступных рабочих узлах. Затем агенты Kubelet на узлах поручают Docker извлечь образы контейнеров из реестра образов и запустить контейнеры.

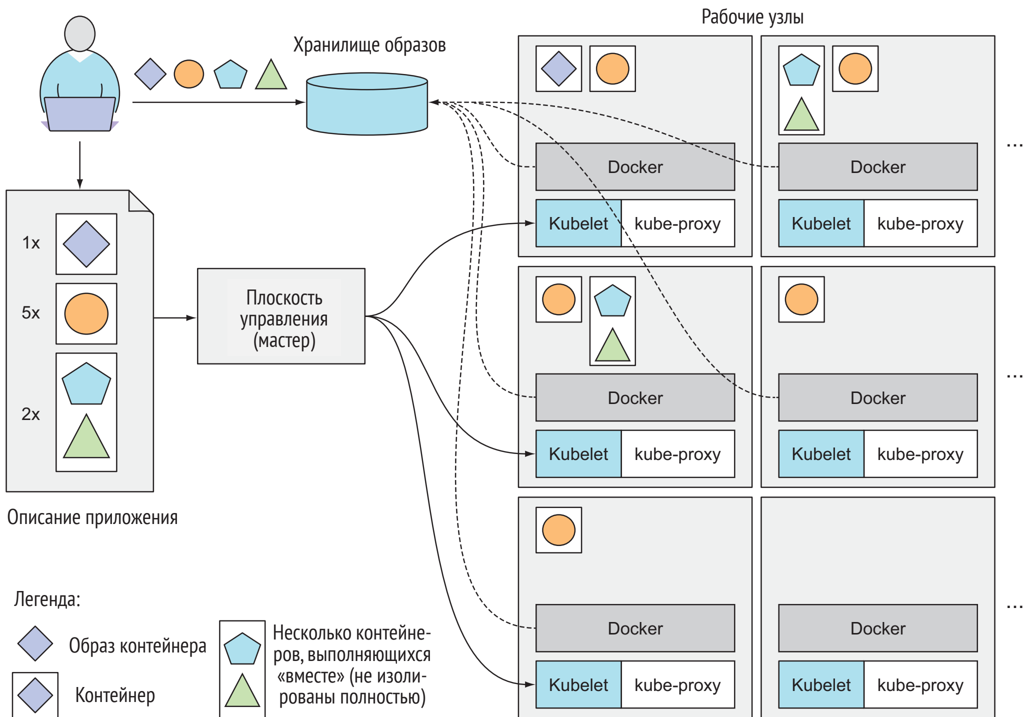


Рис. 1.10. Базовый вид архитектуры Kubernetes и приложения, работающего поверх нее

Обеспечение бесперебойной работы контейнеров

После запуска приложения система Kubernetes постоянно следит за тем, чтобы то, в каком состоянии развернуто, всегда соответствовало указанному вами описанию. Например, если вы укажете, что вы всегда хотите, чтобы работали пять экземпляров веб-сервера, Kubernetes всегда будет поддерживать ровно пять экземпляров. Если один из этих экземпляров перестает работать должным образом, например когда происходит аварийный сбой процесса или он перестает отвечать на запросы, Kubernetes автоматически его перезапустит.

Аналогичным образом, если весь рабочий узел умирает или становится недоступным, Kubernetes выберет новые узлы для всех контейнеров, запущенных на узле, и запустит их на вновь выбранных узлах.

Масштабирование количества копий

Во время работы приложения вы можете решить увеличить или уменьшить количество копий, и Kubernetes соответственно будет запускать дополнительные или остановит лишние. Вы даже можете отказаться от работы по определению оптимального количества копий Kubernetes. Он может автоматически изменять количество на основе реально-временных метрических показателей, таких как загрузка ЦП, потребление памяти, количество запросов в секунду или любого другого показателя, предоставляемого приложением.

Попадание в движущуюся цель

Мы сказали, что системе Kubernetes может потребоваться перемещать ваши контейнеры по кластеру. Это может произойти, если узел, на котором они выполнялись, аварийно завершил свою работу либо они были вытеснены из узла, чтобы освободить место для других контейнеров. Если контейнер предоставляет службу внешним клиентам или другим контейнерам, работающим в кластере, то каким образом они могут правильно использовать контейнер, если он постоянно перемещается по кластеру? И как клиенты могут подключаться к контейнерам, предоставляющим службу, когда эти контейнеры реплицированы и разбросаны по всему кластеру?

Для того чтобы клиенты могли легко находить контейнеры, обеспечивающие определенную службу, вы можете сообщить Kubernetes, какие контейнеры предоставляют одну и ту же службу, и Kubernetes обеспечит доступ ко всем контейнерам по единому статическому IP-адресу и предоставит доступ к этому адресу всем приложениям, работающим в кластере. Это делается через переменные среды, но клиенты могут также обращаться к IP-службам через старый добрый DNS. Прокси-сервер kube, kube-proxy, будет обеспечивать балансировку нагрузки подключений к службе во всех контейнерах, предоставляющих службу. IP-адрес службы остается постоянным, поэтому клиенты всегда могут подключаться к ее контейнерам, даже если они перемещаются по кластеру.

1.3.5 Преимущества использования Kubernetes

Если у вас Kubernetes развернут на всех серверах, то системным администраторам больше не нужно заниматься развертыванием приложений. Поскольку контейнерезированное приложение уже содержит все необходимое для запуска, системным администраторам не нужно ничего устанавливать для развертывания и запуска приложения. На любом узле, где развернута система Kubernetes, Kubernetes может запустить приложение немедленно без помощи системных администраторов.

Упрощение развертывания приложений

Поскольку Kubernetes обеспечивает доступ ко всем своим рабочим узлам как к единой платформе развертывания, разработчики приложений могут начать развертывание приложений самостоятельно, и им не нужно ничего знать о серверах, из которых состоит кластер.

По сути, все узлы теперь представляют собой единую группу вычислительных ресурсов, которые ждут, когда приложения будут их потреблять. Разработчик обычно не заботится о том, на каком сервере работает приложение, если сервер может предоставить приложению достаточный объем системных ресурсов.

Существуют случаи, когда разработчик заботится о том, на каком оборудовании должно работать приложение. Если узлы неоднородны, то вы найдете случаи, когда требуется, чтобы некоторые приложения запускались на узлах с определенными возможностями и запускали другие приложения на других. Например, одно из ваших приложений может потребовать запуска в системе с твердотельными накопителями (SSD) вместо жестких дисков, в то время как другие приложения отлично работают на жестких дисках. В таких случаях вы, очевидно, хотите убедиться, что конкретное приложение всегда запланировано на узле с SSD.

Без использования Kubernetes системный администратор выберет один конкретный узел с SSD и развернет там приложение. Но при использовании Kubernetes вместо выбора конкретного узла, на котором должно выполняться приложение, уместнее поручить Kubernetes выбирать только узлы с SSD. Вы узнаете, как это делать, в главе 3.

Повышение эффективности задействования оборудования

Настроив систему Kubernetes на серверах и используя ее для запуска приложений, а не вручную, вы отделили свое приложение от инфраструктуры. Когда вы поручаете Kubernetes запустить приложение, вы позволяете ей выбрать наиболее подходящий узел для запуска приложения на основе описания потребностей приложения в ресурсах и доступных ресурсов на каждом узле.

Используя контейнеры и не привязывая приложение к определенному узлу в кластере, вы позволяете приложению свободно перемещаться по кластеру в любое время, поэтому различные компоненты приложения, работающие в кластере, могут быть смешаны и приведены к соответствию, чтобы быть плотно упакованными в узлах кластера. Это гарантирует, что аппаратные ресурсы узла используются как можно лучше.

Возможность перемещать приложения по кластеру в любое время позволяет Kubernetes использовать инфраструктуру гораздо лучше, чем то, что можно достичь вручную. Люди не умеют находить оптимальные комбинации, в особенности когда количество всех возможных вариантов огромно, например когда у вас много компонентов приложения и много серверных узлов, на которых они могут быть развернуты. Очевидно, что компьютеры могут выполнять эту работу намного лучше и быстрее, чем люди.

Проверка здоровья и самолечение

Наличие системы, позволяющей перемещать приложение по кластеру в любое время, также полезно в случае аварийных сбоев сервера. По мере увеличения размера кластера вы будете все чаще сталкиваться с неисправными компонентами компьютера.

Kubernetes отслеживает компоненты приложения и узлы, на которых они выполняются, и автоматически переносит их на другие узлы в случае аварийного сбоя узла. Это освобождает системных администраторов от необходимости вручную переносить компоненты приложения и позволяет этой группе немедленно сосредоточиваться на исправлении самого узла и возвращении его в пул доступных аппаратных ресурсов, а не на перемещении приложения.

Если ваша инфраструктура имеет достаточно свободных ресурсов, чтобы позволить нормальную работу системы даже без аварийно вышедшего из строя узла, системным администраторам и не нужно реагировать на аварийный сбой сразу же, например в 3 часа ночи, они могут спать спокойно и разбираться с вышедшим из строя узлом в обычные рабочие часы.

Автоматическое масштабирование

Использование Kubernetes для управления развернутыми приложениями также означает, что системным администраторам не нужно постоянно следить за нагрузкой отдельных приложений, чтобы реагировать на внезапные скачки нагрузки. Как уже упоминалось ранее, системе Kubernetes можно поручить контролировать ресурсы, используемые каждым приложением, и продолжать регулировать количество запущенных экземпляров каждого приложения.

Если Kubernetes работает в облачной инфраструктуре, где добавлять дополнительные узлы так же просто, как запрашивать их через API поставщика облака, Kubernetes даже может автоматически масштабировать размер всего кластера вверх или вниз в зависимости от потребностей развернутых приложений.

Упрощение разработки приложений

Функции, описанные в предыдущем разделе, в основном приносят пользу системным администраторам. Но как насчет разработчиков? Разве Kubernetes не приносит ничего и к их столу? Безусловно, да.

Если вернуться к тому факту, что приложения работают в одной среде как во время разработки, так и в производстве, это оказывает большое влияние на обнаружение ошибок. Мы все согласны с тем, что чем раньше вы обнаружите ошибку, тем легче ее исправить, и ее исправление требует меньше работы. Именно разработчики вносят исправления, а это означает для них меньше работы.

Далее следует упомянуть тот факт, что разработчикам не нужно реализовывать функционал, которым они обычно занимались. Он включает обнаружение служб и/или узлов в кластеризованном приложении. Kubernetes делает

это вместо приложения. Как правило, приложение должно только отыскивать определенные переменные среды или выполнять поиск в DNS. Если этого недостаточно, приложение может запросить сервер API Kubernetes напрямую, чтобы получить эту и/или другую информацию. Подобного рода запрос сервера API Kubernetes может даже сэкономить усилия разработчиков по необходимой реализации сложных механизмов, таких как выбор лидера.

В качестве последнего примера того, что Kubernetes приносит к столу, вы также должны рассмотреть увеличение доверия, которое почувствуют разработчики, зная, что когда новая версия их приложения будет развернута, Kubernetes может автоматически обнаружить, является ли новая версия плохой, и немедленно остановить ее выкладку. Это повышение доверия обычно ускоряет непрерывную доставку приложений, что приносит пользу всей организации.

1.4 Резюме

В этой вводной главе вы увидели то, насколько приложения изменились в последние годы и что теперь они могут быть сложнее в развертывании и управлении. Мы представили систему Kubernetes и показали, как эта система вместе с Docker и другими контейнерными платформами помогает развертывать и управлять приложениями и инфраструктурой, в которой они работают. Вы узнали, что:

- монолитные приложения проще развертывать, но сложнее поддерживать с течением времени, а иногда и невозможно масштабировать;
- архитектуры приложений на основе микросервисов упрощают разработку каждого компонента, но их сложнее развертывать и настраивать для работы в качестве единой системы;
- контейнеры Linux предоставляют те же преимущества, что и виртуальные машины, но гораздо облегченнее и обеспечивают намного лучшую степень задействованности оборудования;
- платформа Docker улучшила существующие контейнерные технологии Linux, позволив упростить и ускорить создание контейнеризированных приложений вместе с окружением их операционных систем;
- система Kubernetes обеспечивает доступ ко всему дата-центру как к единому вычислительному ресурсу для запуска приложений;
- разработчики могут развертывать приложения с помощью Kubernetes без помощи сисадминов;
- администраторы могут лучше спать, имея систему Kubernetes, которая автоматически будет заниматься узлами, аварийно завершившими свою работу.

В следующей главе вы закатаете рукава и займетесь конкретной работой: созданием приложения и его запуском в Docker, а затем в Kubernetes.

Глава 2

Первые шаги с Docker и Kubernetes

Эта глава посвящена:

- созданию, запуску и распространению образа контейнера с помощью Docker;
- локальному запуску одноузлового кластера Kubernetes;
- настройке кластера Kubernetes в Google Kubernetes Engine;
- настройке и использованию клиента командной строки kubectl;
- развертыванию приложения на Kubernetes и горизонтальному масштабированию.

Прежде чем вы начнете подробно изучать понятия системы Kubernetes, давайте посмотрим, как создать простое приложение, упаковать его в образ контейнера и запустить в управляемом кластере Kubernetes (в Google Kubernetes Engine) или в локальном кластере с одним узлом. Это должно дать вам немного более предметный обзор всей системы Kubernetes и облегчит прохождение следующих нескольких глав, где мы рассмотрим основные строительные блоки и понятия, принятые в Kubernetes.

2.1 Создание, запуск и совместное использование образа контейнера

Как вы уже узнали в предыдущей главе, запуск приложений в Kubernetes требует их упаковки в образы контейнеров. Мы дадим начальные сведения об использовании контейнерной платформы Docker, если вы еще ее не использовали. В следующих нескольких разделах вы:

1) установите Docker и запустите свой первый контейнер «Hello world»;
2) создадите тривиальное приложение Node.js, которое вы позже развернете в Kubernetes;

3) упакуете приложение в образ контейнера, чтобы затем запустить его как изолированный контейнер;
4) запустите контейнер на основе образа;
5) переместите образ в хранилище Docker Hub, чтобы его мог запустить любой пользователь.

2.1.1 Установка Docker и запуск контейнера Hello World

Прежде всего вам нужно установить Docker на вашей машине Linux. Если вы не используете Linux, то необходимо запустить виртуальную машину Linux и запустить Docker на этой виртуальной машине. Если вы используете Mac или Windows и устанавливаете Docker в соответствии с инструкциями, то Docker настроит для вас виртуальную машину и запустит демон Docker внутри этой виртуальной машины. Исполняемый файл клиента Docker будет доступен в хостовой ОС и будет взаимодействовать с демоном внутри виртуальной машины.

Для того чтобы установить Docker, следуйте инструкциям на странице http://docs.docker.com/engine/installation/ для конкретной операционной системы. После завершения установки можно для выполнения различных команд Docker использовать исполняемый файл клиента Docker. Например, можно попробовать загрузить и запустить существующий образ из Docker Hub, общедоступного хранилища Docker, который содержит готовые к использованию образы контейнеров для многих известных пакетов программного обеспечения. Одним из них является образ `busybox`, который вы будете использовать для запуска простой команды `echo "Hello world"`.

Запуск контейнера hello world

Если вы незнакомы с `busybox`, то это одиночный исполняемый файл, который сочетает в себе многие стандартные инструменты командной строки UNIX, такие как `echo`, `ls`, `gzip` и т. д. Вместо образа `busybox` можно также использовать любой другой полноценный системный образ контейнера, например Fedora, Ubuntu или другие подобные образы, если он включает исполняемый файл `echo`.

Как запустить образ `busybox`? Вам не нужно ничего скачивать или устанавливать. Используйте команду `docker run` и укажите, какой образ загрузить и запустить и (факультативно) какую команду выполнить, как показано в следующем ниже листинге.

Листинг 2.1. Запуск контейнера Hello world с помощью Docker

```
$ docker run busybox echo "Hello world"
Unable to find image 'busybox:latest' locally
latest: Pulling from docker.io/busybox
9a163e0b8d13: Pull complete
fef924a0204a: Pull complete
```

```
Digest: sha256:97473e34e311e6c1b3f61f2a721d038d1e5eef17d98d1353a513007cf46ca6bd
Status: Downloaded newer image for docker.io/busybox:latest
Hello world
```

Это не выглядит как-то впечатляюще, но если вы учтете, что все «приложение» было скачано и выполнено с помощью одной команды, без необходимости устанавливать это приложение или что-либо еще, то согласитесь, что это потрясающе. В вашем случае приложение было одиночным исполняемым файлом (busybox), но оно могло быть невероятно сложным приложением со множеством зависимостей. Весь процесс настройки и запуска приложения был бы точно таким же. Важно также, что приложение было выполнено внутри контейнера, полностью изолированного от всех других процессов, запущенных на вашей машине.

Что происходит за кулисами

На рис. 2.1 показано, что именно произошло при выполнении команды `docker run`. Сначала платформа Docker проверила, присутствует ли на вашей локальной машине образ `busybox:latest`. Он отсутствовал, поэтому Docker извлекла его из хранилища Docker Hub в http://docker.io. После скачивания образа на вашу машину Docker создала из этого образа контейнер и запустила в нем команду. Команда `echo` напечатала текст в STDOUT, а затем процесс завершился, и контейнер был остановлен.

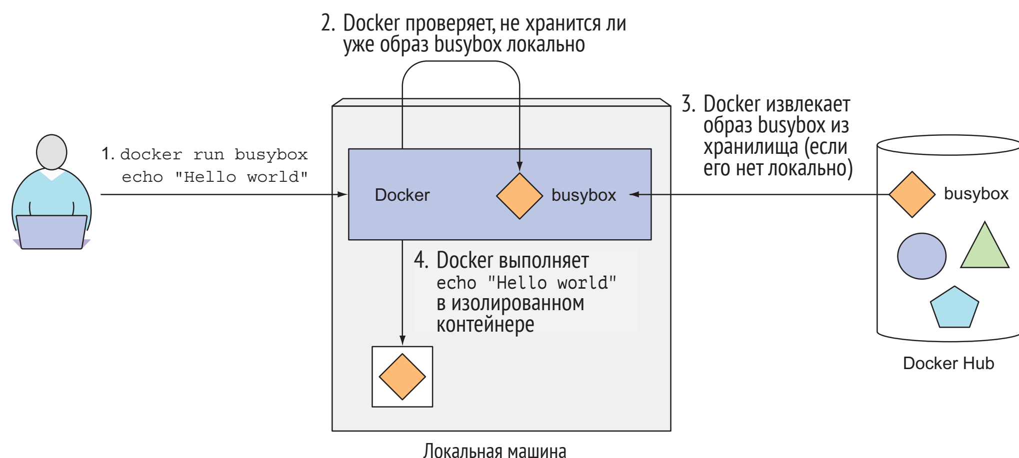


Рис. 2.1. Выполнение `echo "Hello world"` в контейнере на основе образа busybox контейнера

Запуск других образов

Запуск других существующих образов контейнеров аналогичен запуску образа busybox. На самом деле это часто даже проще, потому что вам обычно не нужно указывать, какую команду выполнить, как вы это сделали в примере (`echo "Hello world"`). Команда, которая должна быть выполнена, обычно внедрена в сам образ, но вы можете переопределить ее, если хотите. После поиска

или просмотра общедоступных образов на http://hub.docker.com или другом публичном хранилище вы поручаете Docker запустить образ, как ниже:

```
$ docker run <образ>
```

Управление версиями образов контейнеров

Все пакеты программного обеспечения обновляются, поэтому обычно существует более одной версии пакета. Docker поддерживает под одним именем несколько версий или вариантов одного и того же образа. Каждый вариант должен иметь уникальный тег. При обращении к образам без явного указания тега Docker будет считать, что вы ссылаетесь на так называемый *последний* (latest) тег. Для того чтобы запустить другую версию образа, можно указать тег вместе с именем образа следующим образом:

```
$ docker run <образ>:<тег>
```

2.1.2 Создание тривиального приложения Node.js

Теперь, когда у вас есть работающая настройка платформы Docker, вы создадите приложение. Вы построите тривиальное веб-приложение Node.js и упакуете его в образ контейнера. Приложение будет принимать HTTP-запросы и отвечать хостнеймом машины, на которой оно выполняется. Таким образом, вы увидите, что приложение, запущенное внутри контейнера, видит собственный хостнейм, а не имя хостовой машины, даже если оно работает на хосте, как и любой другой процесс. Это будет полезно позже при развертывании приложения на Kubernetes и его масштабировании вширь (горизонтальном масштабировании, то есть запуске нескольких экземпляров приложения). Вы увидите, что ваши HTTP-запросы попадают в разные экземпляры приложения.

Ваше приложение будет состоять из одного файла под именем app.js с содержимым, показанным в следующем ниже листинге.

Листинг 2.2. Простое приложение Node.js: app.js

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");

var handler = function(request, response) {
  console.log("Received request from " + request.connection.remoteAddress);
  response.writeHead(200);
  response.end("You've hit " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

То, что этот код делает, должно быть понятно. Он запускает HTTP-сервер на порту `8080`. На каждый запрос сервер откликается кодом состояния HTTP `200 OK` и текстом `"You've hit <hostname>"`. Обработчик запросов также записывает клиентский IP-адрес в стандартный вывод, который понадобится позже.

> **ПРИМЕЧАНИЕ.** Возвращаемый хостнейм имя является фактическим хостнеймом сервера, а не именем, которое клиент отправляет в заголовке Host HTTP-запроса.

Теперь можно загрузить и установить Node.js и протестировать приложение напрямую, но это не обязательно, так как вы будете использовать Docker для упаковки приложения в образ контейнера и его запуска в любом месте без необходимости что-либо загружать или устанавливать (кроме Docker, который должен быть установлен на машине, на которой вы хотите запустить образ).

Выбор базового образа

Вы можете поинтересоваться, почему мы выбрали этот специфический образ в качестве базового. Поскольку вашим приложением является приложение Node.js, вам нужно, чтобы ваш образ содержал двоичный исполняемый файл `node` для запуска приложения. Вы могли бы использовать любой образ, который содержит этот двоичный файл, или даже использовать базовый образ дистрибутива Linux, такой как `fedora` или `ubuntu`, и установить Node.js в контейнер во время сборки образа. Но поскольку образ node создан специально для выполнения приложений Node.js и включает в себя все, что вам нужно, чтобы запустить приложение, вы будете использовать его в качестве базового образа.

2.1.3 Создание файла Dockerfile для образа

Для того чтобы упаковать приложение в образ, сначала необходимо создать файл Dockerfile, содержащий список инструкций, которые Docker будет выполнять при сборке образа. Файл Dockerfile должен находиться в том же каталоге, что и файл app.js, и должен содержать команды, показанные в следующем ниже листинге.

Листинг 2.3. Файл Dockerfile для создания образа контейнера вашего приложения

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

Строка `FROM` определяет образ контейнера, который вы будете использовать в качестве отправной точки (базовый образ, поверх которого вы будете надстраивать). В вашем случае вы используете контейнерный образ `node`, тег 7. Во второй строке вы добавляете файл app.js из вашего локального каталога в кор-

невой каталог образа под тем же именем (app.js). Наконец, в третьей строке вы определяете, какая команда должна быть выполнена, когда кто-то запускает образ. В вашем случае командой является `node app.js`.

2.1.4 Создание контейнерого образа

Теперь, когда у вас есть файл Dockerfile и файл app.js, у вас есть все необходимое для создания образа. Для того чтобы собрать его, выполните следующую ниже команду Docker:

```
$ docker build -t kubia .
```

На рис. 2.2 показано, что происходит в процессе сборки. Вы поручаете Docker создать образ с именем `kubia` на основе содержимого текущего каталога (обратите внимание на точку в конце команды `build`). Docker будет искать файл Dockerfile в каталоге и строить образ на основе инструкций в этом файле.

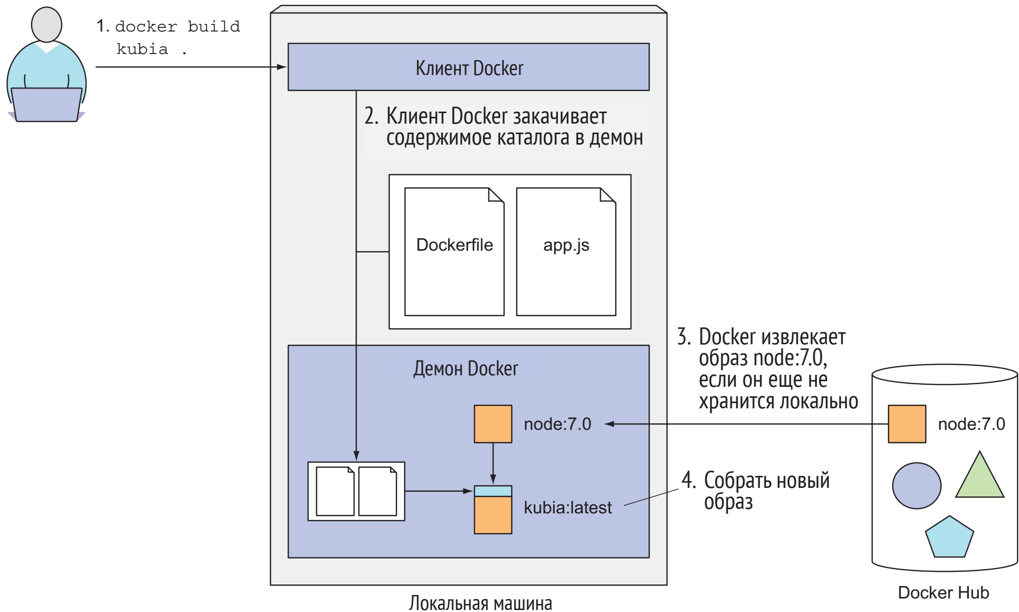


Рис. 2.2. Создание нового образа контейнера из Dockerfile

Как создается образ

Процесс сборки не выполняется клиентом Docker. Вместо этого содержимое всего каталога закачивается в демон Docker, и там создается образ. Клиент и демон не обязательно должны находиться на одной машине. Если вы используете Docker не в Linux'овской ОС, то клиент находится в хостовой ОС, но демон работает внутри виртуальной машины. Все файлы в каталоге сборки закачиваются в демон, поэтому, если он содержит много больших файлов и демон не работает локально, закачка может занять много времени.

> **СОВЕТ.** Не включайте ненужные файлы в каталог сборки, так как они замедляют процесс сборки, в особенности если демон Docker находится на удаленной машине.

Во время процесса сборки сначала Docker извлекает базовый образ (node:7) из общедоступного хранилища образов (Docker Hub), если только образ уже не был извлечен и не сохранен на вашей машине.

Слои образа

Образ не является единым, большим двоичным блоком, а состоит из нескольких слоев, на которые вы, возможно, уже обратили внимание при выполнении примера busybox (выводилось несколько строк Pull complete – по одной для каждого). Разные образы могут иметь несколько слоев, что делает хранение и передачу образов гораздо эффективнее. Например, если вы создаете несколько образов на основе одного и того же базового образа (например, node:7 в примере), то все слои, содержащие базовый образ, будут сохранены всего один раз. Кроме того, при выгрузке образа платформа Docker загружает каждый слой по отдельности. Несколько слоев могут уже храниться на вашей машине, так что Docker будет скачивать только те, которые отсутствуют.

Вы можете подумать, что каждый файл Dockerfile создает только один новый слой, но это не так. При сборке образа создается новый слой для каждой отдельной команды в Dockerfile. Во время сборки вашего образа после извлечения всех слоев базового образа платформа Docker создаст новый слой поверх них и добавит в него файл app.js. Затем она создаст еще один слой, который будет указывать команду, которая должна быть выполнена при запуске образа. Этот последний слой будет помечен как `kubia:latest`. Это показано на рис. 2.3, также объясняющем, как другой образ под названием `other:latest` будет использовать те же самые слои образа Node.js, что и ваш собственный образ.

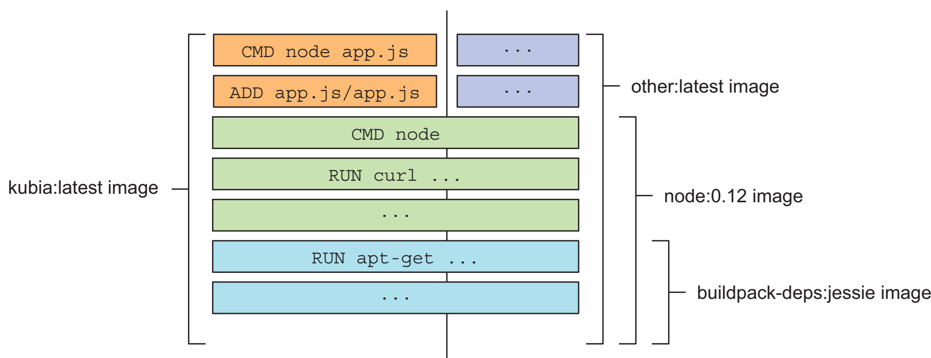


Рис. 2.3. Образы контейнеров состоят из слоев, которые могут совместно использоваться разными образами

После завершения процесса сборки новый образ будет сохранен локально. Вы можете увидеть это, поручив платформе Docker вывести список всех локально хранящихся образов, как показано в следующем ниже листинге.

Листинг 2.4. Вывод списка локально хранящихся образов

```
$ docker images
REPOSITORY     TAG        IMAGE ID        CREATED          VIRTUAL SIZE
kubia          latest     d30ecc7419e7    1 minute ago     637.1 MB
...
```

Сравнение процессов сборки образов с помощью dockerfile и вручную

Файлы Dockerfile – это обычный способ сборки образов контейнеров с помощью Docker, но вы также можете создать образ вручную, запустив контейнер из существующего образа, выполнив команды в контейнере, выйдя из контейнера и зафиксировав конечное состояние как новый образ (отправив коммит). Это именно то, что происходит, когда вы выполняете сборку из Dockerfile, но она выполняется автоматически и повторима, что позволяет вносить изменения в Dockerfile и собирать образ повторно в любое время, без необходимости вручную перенабирать все команды снова.

2.1.5 Запуск образа контейнера

Теперь образ можно запустить с помощью следующей ниже команды:

```
$ docker run --name kubia-container -p 8080:8080 -d kubia
```

Она поручает платформе Docker запустить новый контейнер с именем `kubia-container` из образа `kubia`. Контейнер будет отсоединен от консоли (флаг `-d`), имея в виду, что он будет работать в фоновом режиме. Порт `8080` на локальной машине будет увязан с портом `8080` внутри контейнера (параметр `-p 8080:8080`), так что вы можете получить доступ к приложению через http://localhost:8080.

Если вы не выполняете демон Docker на вашей локальной машине (при использовании Mac или Windows демон работает внутри виртуальной машины), то вам вместо localhost нужно использовать сетевое имя или IP виртуальной машины, на которой работает демон. Вы можете узнать его через переменную среды `DOCKER_HOST`.

Доступ к приложению

Теперь попробуйте получить доступ к приложению на http://localhost:8080 (при необходимости замените localhost сетевым именем или IP-адресом хоста Docker):

```
$ curl localhost:8080
You've hit 44d76963e8e1
```

Это отклик вашего приложения. Ваше крошечное приложение теперь работает внутри контейнера, изолированного от всего остального. Как вы можете

видеть, в качестве своего хостнейма оно возвращает `44d76963e8e1`, а не фактический хостнейм вашей хост-машины. Приведенное выше шестнадцатеричное число – это идентификатор контейнера Docker.

Список всех запущенных контейнеров

В следующем ниже листинге давайте выведем список всех запущенных контейнеров, чтобы вы могли изучить список (я отредактировал результат, чтобы сделать его более читаемым, – представьте, что последние две строки являются продолжением первых двух).

Листинг 2.5. Список запущенных контейнеров

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        ...
44d76963e8e1   kubia:latest   "/bin/sh -c 'node ap     6 minutes ago  ...

...  STATUS             PORTS                      NAMES
...  Up 6 minutes       0.0.0.0:8080->8080/tcp     kubia-container
```

Выполняется один контейнер. По каждому контейнеру Docker печатает его идентификатор и имя, образ, используемый для выполнения контейнера, и команду, исполняемую внутри контейнера.

Получение дополнительной информации о контейнере

Команда `docker ps` показывает только самые основные сведения о контейнерах. Для просмотра дополнительных сведений можно применить команду `docker inspect`:

```
$ docker inspect kubia-container
```

Docker распечатает длинный документ JSON, содержащий низкоуровневую информацию о контейнере.

2.1.6 Исследование работающего контейнера изнутри

Что делать, если вы хотите увидеть, как выглядит среда внутри контейнера? Поскольку в одном контейнере может выполняться несколько процессов, вы всегда можете запустить в нем дополнительный процесс, чтобы увидеть, что находится внутри. Можно даже запустить оболочку (шелл), при условии что двоичный исполняемый файл оболочки доступен в образе.

Запуск оболочки внутри существующего контейнера

Образ Node.js, на котором вы основываете свой образ, содержит оболочку bash, поэтому ее можно запустить в контейнере следующим образом:

```
$ docker exec -it kubia-container bash
```

Эта команда запустит bash внутри существующего контейнера `kubia-container`. Процесс bash будет иметь те же пространства имен Linux, что и главный процесс контейнера. Это позволяет исследовать контейнер изнутри и увидеть, как Node.js и ваше приложение видят систему во время выполнения внутри контейнера. Параметр `-it` является сокращением для двух параметров:

- `-i` убеждается, что STDIN держится открытым. Это необходимо для ввода команд в оболочку;
- `-t` выделяет псевдотерминал (TTY).

Если вы хотите использовать оболочку, как вы привыкли это делать, то вам нужны оба. (Если опустить первый, вы не сможете вводить какие-либо команды, а если опустить второй, то командная строка не будет отображаться, и некоторые команды будут жаловаться на то, что переменная `TERM` не задана.)

Исследование контейнера изнутри

Давайте посмотрим, как использовать оболочку, в следующем далее листинге, чтобы увидеть процессы, запущенные в контейнере.

Листинг 2.6. Вывод списка процессов внутри контейнера

```
root@44d76963e8e1:/# ps aux
USER  PID  %CPU  %MEM    VSZ    RSS  TTY  STAT  START  TIME  COMMAND
root    1   0.0   0.1  676380  16504  ?    Sl    12:31  0:00  node app.js
root   10   0.0   0.0   20216   1924  ?    Ss    12:31  0:00  bash
root   19   0.0   0.0   17492   1136  ?    R+    12:38  0:00  ps aux
```

Вы видите только три процесса. Вы не видите никаких других процессов из хостовой ОС.

Понимание того, что процессы в контейнере выполняются в хостовой операционной системе

Если теперь открыть другой терминал и вывести список процессов в самой хостовой ОС, то среди всех других хостовых процессов вы также увидите процессы, запущенные в контейнере, как показано в листинге 2.7.

> **ПРИМЕЧАНИЕ.** Если вы используете Mac или Windows, то для того, чтобы видеть эти процессы, вам необходимо войти в виртуальную машину, где работает демон Docker.

Листинг 2.7. Процессы контейнера, работающие в хостовой ОС

```
$ ps aux | grep app.js
USER  PID  %CPU  %MEM    VSZ    RSS  TTY  STAT  START  TIME  COMMAND
root  382   0.0   0.1  676380  16504  ?    Sl    12:31  0:00  node app.js
```

Это доказывает, что процессы, запущенные в контейнере, выполняются в хостовой ОС. Если у вас острый глаз, то вы, возможно, заметили, что процессы имеют разные идентификаторы внутри контейнера по сравнению с хостом. Контейнер использует собственное Linux-пространство имен PID и имеет полностью изолированное дерево процессов с собственной последовательностью номеров PID.

Файловая система контейнера также изолирована

Подобно изолированному дереву процессов, каждый контейнер также имеет изолированную файловую систему. Список содержимого корневого каталога внутри контейнера будет показывать только файлы в контейнере и будет включать все файлы, которые находятся в образе, а также все файлы, созданные во время работы контейнера (файлы журналов и аналогичные), как показано в следующем ниже листинге.

Листинг 2.8. Контейнер имеет собственную полную файловую систему

```
root@44d76963e8e1:/# ls /
app.js  boot  etc   lib    media  opt   root  sbin  sys  usr
bin     dev   home  lib64  mnt    proc  run   srv   tmp  var
```

Он содержит файл app.js и другие системные каталоги, являющиеся частью базового используемого образа `node:7`. Для того чтобы выйти из контейнера, следует выйти из оболочки, выполнив команду `exit`, и вы вернетесь на свою хост-машину (например, выйдя из сеанса ssh).

> **СОВЕТ.** Входить в подобный работающий контейнер полезно при отладке приложения, запущенного в контейнере. Когда что-то не так, первое, что вы захотите изучить, будет фактическое состояние системы, которое видит ваше приложение. Имейте в виду, что приложение будет видеть не только свою собственную уникальную файловую систему, но и процессы, пользователей, хостнейм и сетевые интерфейсы.

2.1.7 Остановка и удаление контейнера

Для того чтобы остановить приложение, поручите Docker остановить контейнер `kubia-container`:

```
$ docker stop kubia-container
```

Это остановит главный процесс, запущенный в контейнере, и, следовательно, остановит контейнер, поскольку никакие другие процессы не выполняются внутри контейнера. Сам контейнер все еще существует, и вы можете увидеть его с помощью команды `docker ps -a`. Параметр `-a` выводит на печать все контейнеры, запущенные и остановленные. Для того чтобы действительно удалить контейнер, необходимо удалить его с помощью команды `docker rm`:

```
$ docker rm kubia-container
```

Это приведет к удалению контейнера. Все его содержимое удаляется, и он не может быть запущен снова.

2.1.8 Отправка образа в хранилище образов

Созданный вами образ пока доступен только на вашей локальной машине. Для того чтобы разрешить запуск на любой другой машине, необходимо отправить образ во внешнее хранилище образов. Для простоты вы не будете настраивать приватное хранилище образов и вместо этого будете отправлять образ в хранилище Docker Hub (http://hub.docker.com), который является одним из общедоступных хранилищ. Другими широко используемыми хранилищами подобного рода являются Quay.io и хранилище контейнеров Google Container Registry.

Перед этим необходимо перетегировать образ в соответствии с правилами хранилища Docker Hub. Docker Hub позволяет загрузить образ, если имя репозитория образа начинается с Docker Hub ID. Docker Hub ID можно создать, зарегистрировавшись по адресу http://hub.docker.com. В последующих примерах я буду использовать свой собственный ID (`luksa`). Пожалуйста, поменяйте каждый случай его использования на свой собственный идентификатор.

Тегирование образа дополнительным тегом

После того как вы узнали свой код, вы готовы переименовать свой образ, в настоящее время помеченный тегом `kubia`, на `luksa/kubia` (поменяйте `luksa` на свой собственный Docker Hub ID):

```
$ docker tag kubia luksa/kubia
```

Эта команда не переименовывает тег. Вместо этого она создает дополнительный тег для того же самого образа. Вы можете подтвердить это, выведя список образов, хранящихся в вашей системе. Для этого надо выполнить команду `docker images`, как показано в следующем ниже листинге.

Листинг 2.9. Образ контейнера может иметь несколько тегов

```
$ docker images | head
REPOSITORY      TAG     IMAGE ID      CREATED            VIRTUAL SIZE
luksa/kubia     latest  d30ecc7419e7  About an hour ago  654.5 MB
kubia           latest  d30ecc7419e7  About an hour ago  654.5 MB
docker.io/node  7.0     04c0ca2a8dad  2 days ago         654.5 MB
...
```

Как вы можете видеть, оба тега – `kubia` и `luksa/kubia` – указывают на один и тот же идентификатор образа, поэтому они на самом деле являются одним-единственным образом с двумя тегами.

Передача образа в хранилище docker hub

Прежде чем вы сможете отправить образ в хранилище Docker Hub, необходимо войти в систему под своим ID пользователя с помощью команды `docker login`. После того как вы вошли в систему, вы можете, наконец, отправить образ `вашID/kubia` в хранилище Docker Hub, как показано ниже:

```
$ docker push luksa/kubia
```

Запуск образа на другой машине

После завершения отправки образа в хранилище Docker Hub образ будет доступен всем. Теперь образ можно запустить на любой машине, на которой выполняется Docker. Для этого надо выполнить следующую ниже команду:

```
$ docker run -p 8080:8080 -d luksa/kubia
```

Проще уже просто невозможно. И самое лучшее в этом то, что ваше приложение будет иметь точно такую же среду всякий раз и везде, где оно выполняется. Если оно отлично работает на вашей машине, оно должно работать также на любой другой машине Linux. Нет необходимости беспокоиться о том, установлено на хост-машине ПО Node.js или нет. На самом деле даже если это так, ваше приложение использовать его не будет, потому что оно будет использовать то, которое установлено внутри образа.

2.2 Настройка кластера Kubernetes

Теперь, когда приложение упаковано в образ контейнера и доступно через реестр Docker Hub, его можно развернуть в кластере Kubernetes, а не запускать непосредственно в Docker. Но сначала нужно настроить сам кластер.

Создание полноценного многоузлового кластера Kubernetes – задача непростая, в особенности если вы не разбираетесь в администрировании Linux и сетей. Правильная установка Kubernetes охватывает несколько физических или виртуальных машин и требует правильной настройки сети, чтобы все контейнеры, работающие в кластере Kubernetes, могли подключаться друг к другу через одно и то же прозрачное пространство сетевого взаимодействия.

Существует длинный список методов установки кластера Kubernetes. Эти методы подробно описаны в документации на http://kubernetes.io. Мы не будем перечислять их все здесь, потому что этот список все время меняется, но система Kubernetes может быть запущена на вашей локальной машине, предназначенной для разработки, на кластере машин вашей собственной организации, на облачных провайдерах, предоставляющих виртуальные машины (вычислительной системе Google Compute Engine, облачной службе Amazon EC2, облаке Microsoft Azure и т. д.), или с помощью управляемого кластера Kubernetes, такого как система управления и оркестровки контейнеров Google Kubernetes Engine (ранее называвшаяся Google Container Engine).

В этой главе для получения практического опыта в запуске кластера Kubernetes мы рассмотрим два простых варианта. Вы увидите, как запускать одноузловой кластер Kubernetes на вашей локальной машине и как получать доступ к размещенному кластеру, работающему на Google Kubernetes Engine (GKE).

Третий вариант, который охватывает установку кластера с помощью инструмента `kubeadm`, объясняется в приложении B. Приведенные там инструкции показывают, как настроить трехузловой кластер Kubernetes с помощью виртуальных машин, но я предлагаю вам попробовать это только после прочтения первых 11 глав книги.

Еще один вариант – установить Kubernetes на веб-службах Amazon AWS (Amazon Web Services). Для этого вы можете взглянуть на инструмент `kops`, который надстроен поверх `kubeadm`, упомянутого в предыдущем абзаце, и доступен на http://github.com/kubernetes/kops. Он поможет вам развернуть высокодоступный кластер Kubernetes боевого уровня в AWS и в конечном итоге поддерживать другие платформы (Google Kubernetes Engine, VMware, vSphere и т. д.).

2.2.1 Запуск локального одноузлового кластера Kubernetes с помощью minikube

Самый простой и быстрый путь к полностью функционирующему кластеру Kubernetes – использовать инструмент Minikube. Minikube – это инструмент, который настраивает одноузловой кластер. Такой кластер отлично подходит как для тестирования системы Kubernetes, так и для разработки приложений локально.

Хотя мы не можем показать некоторые функциональные средства Kubernetes, связанные с управлением приложениями на нескольких узлах, одноузлового кластера должно быть достаточно для изучения большинства тем, обсуждаемых в этой книге.

Установка инструмента minikube

Minikube – это одиночный двоичный файл, который необходимо скачать и поместить на свой домашний путь. Этот инструмент имеется для OS X, Linux и Windows. Для того чтобы установить его, для начала лучшего всего перейти в репозиторий Minikube на GitHub (http://github.com/kubernetes/minikube) и следовать приведенным там инструкциям.

Например, в OSX и Linux Minikube можно загрузить и настроить с помощью одной команды. Вот как выглядит команда для OSX:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/
➥  v0.23.0/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube
➥  /usr/local/bin/
```

В Linux скачивается другой выпуск (замените «darwin» на «linux» в URL-адресе). В Windows можно скачать файл вручную, переименовать его в miniku-

be.exe и положить его на свой домашний путь. Minikube запускает Kubernetes внутри виртуальной машины, запущенной через VirtualBox или KVM, поэтому, прежде чем вы сможете запустить кластер Minikube, вам также нужно установить один из них.

Запуск кластера kubernetes с помощью minikube

После того как Minikube установлен локально, можно сразу же запустить кластер Kubernetes. Для этого используется команда, приведенная в следующем ниже листинге.

Листинг 2.10. Запуск виртуальной машины Minikube

```
$ minikube start
Starting local Kubernetes cluster...
Starting VM...
SSH-ing files into VM...
...
Kubectl is now configured to use the cluster.
```

Запуск кластера занимает больше минуты, поэтому не прерывайте выполнение команды до ее завершения.

Установка клиента kubernetes (kubectl)

Для взаимодействия с Kubernetes также необходим клиент CLI `kubectl`. Опять-таки от вас только требуется скачать его и положить на свой домашний путь. Например, последнюю стабильную версию для OS X можно скачать и установить с помощью следующей ниже команды:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release
➥  /$(curl -s https://storage.googleapis.com/kubernetes-release/release
➥  /stable.txt)/bin/darwin/amd64/kubectl
➥  && chmod +x kubectl
➥  && sudo mv kubectl /usr/local/bin/
```

Для того чтобы скачать `kubectl` для Linux или Windows, замените `darwin` в URL-адресе на `linux` или `windows`.

> **ПРИМЕЧАНИЕ.** Если вы будете использовать несколько кластеров Kubernetes (например, Minikube и GKE), обратитесь к приложению A для получения информации о настройке и переключении между различными контекстами `kubectl`.

Проверка работы кластера и обмена информацией с kubectl

Для того чтобы убедиться, что кластер работает, можно использовать команду `kubectl cluster-info`, показанную в следующем ниже листинге.

Листинг 2.11. Вывод информации о кластере

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/proxy/...
kubernetes-dashboard is running at https://192.168.99.100:8443/api/v1/...
```

Она показывает, что кластер поднят. В нем показаны URL-адреса различных компонентов Kubernetes, включая сервер API и веб-консоль.

> **СОВЕТ.** Вы можете выполнить `minikube ssh`, чтобы войти в виртуальную машину Minikube и исследовать ее изнутри. Например, можно посмотреть, какие процессы выполняются на узле.

2.2.2 Использование кластера Kubernetes, предоставляемого как сервис с Google Kubernetes Engine

Если требуется исследовать полноценный многоузловой кластер Kubernetes, то можно использовать управляемый кластер Google Kubernetes Engine (GKE). Тем самым вам не нужно вручную настраивать все узлы и сетевое взаимодействие кластера, что обычно чересчур напрягает тех, кто делает свои первые шаги с Kubernetes. Использование управляемого решения, такого как GKE, гарантирует, что вы не получите неправильно настроенный, не работающий или частично работающий кластер.

Настройка проекта в облаке google и скачивание необходимых клиентских бинарников

Перед настройкой нового кластера Kubernetes необходимо настроить среду GKE. Поскольку данный процесс может измениться, я не буду здесь приводить точные инструкции. Для начала, пожалуйста, последуйте инструкциям на https://cloud.google.com/container-engine/docs/before-you-begin.

Грубо говоря, вся процедура включает в себя:

1) регистрацию аккаунта Google, маловероятно, что у вас его еще нет;
2) создание проекта в консоли Google Cloud Platform;
3) включение биллинга. Для этого требуется информация о вашей кредитной карте, но Google предоставляет 12-месячную бесплатную пробную версию. И они достаточно порядочны в том, что не снимают плату автоматически после окончания бесплатной пробной версии;
4) включение Kubernetes Engine API;
5) скачивание и установка Google Cloud SDK (сюда входит инструмент командной строки `gcloud`, который необходим для создания кластера Kubernetes);
6) установку утилиты командной строки `kubectl` с помощью `gcloud components install kubectl`.

ПРИМЕЧАНИЕ. Некоторые операции (в шаге 2, например) могут занять несколько минут, поэтому пока суть да дело, можно расслабиться и выпить чашечку кофе.

Создание трехузлового кластера kubernetes

После завершения установки можно создать кластер Kubernetes с тремя рабочими узлами. Для этого следует воспользоваться командой, показанной в следующем ниже листинге.

Листинг 2.12. Создание трехузлового кластера в GKE

```
$ gcloud container clusters create kubia --num-nodes 3
➥  --machine-type f1-micro
Creating cluster kubia...done.
Created [https://container.googleapis.com/v1/projects/kubia1-
1227/zones/europe-west1-d/clusters/kubia].
kubeconfig entry generated for kubia.
NAME ZONE MST_VER MASTER_IP TYPE NODE_VER NUM_NODES STATUS
kubia eu-w1d 1.5.3 104.155.92.30 f1-micro 1.5.3 3 RUNNING
```

Теперь у вас должен быть запущенный кластер Kubernetes с тремя рабочими узлами, как показано на рис. 2.4. Вы используете три узла, для того чтобы лучше продемонстрировать функционал, применяемый к нескольким узлам. При желании можно использовать меньшее количество узлов.

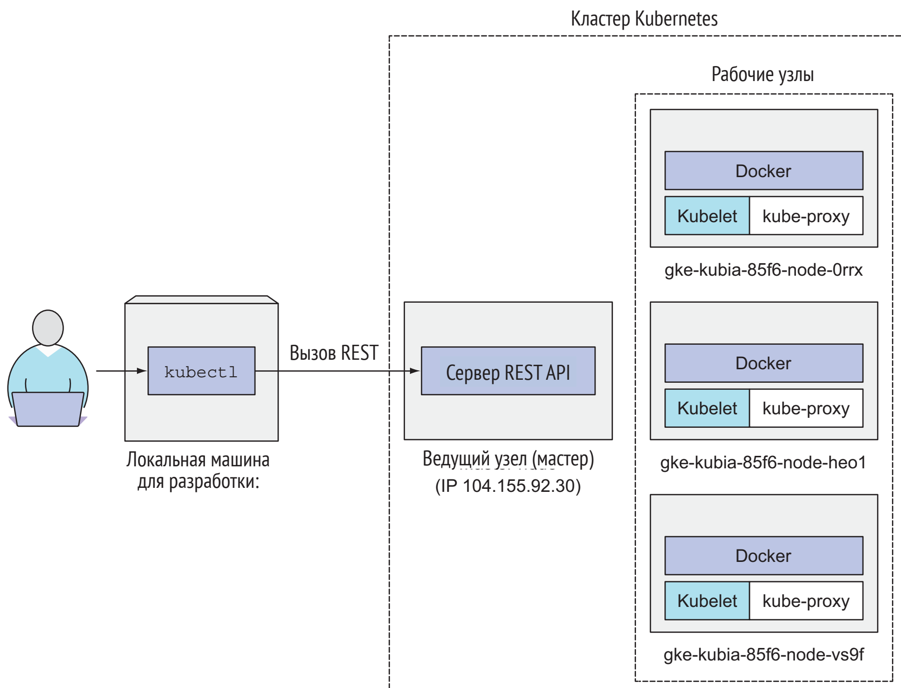


Рис. 2.4. Как происходит взаимодействие с тремя узлами кластера Kubernetes

Общий вид кластера

Для того чтобы дать вам общее представление о том, как выглядит кластер и как с ним взаимодействовать, обратитесь к рис. 2.4. Каждый узел запускает Docker, Kubelet и kube-proxy. Вы будете взаимодействовать с кластером через клиента командной строки `kubectl`, который отправляет запросы REST на сервер API Kubernetes, работающий на ведущем узле.

Проверка рабочего состояния кластера путем вывода списка узлов кластера

Теперь вы воспользуетесь командой `kubectl` для вывода списка всех узлов кластера, как показано в следующем ниже листинге.

Листинг 2.13. Вывод списка узлов кластера с помощью kubectl

```
$ kubectl get nodes
NAME                     STATUS  AGE  VERSION
gke-kubia-85f6-node-0rrx Ready   1m   v1.5.3
gke-kubia-85f6-node-heo1 Ready   1m   v1.5.3
gke-kubia-85f6-node-vs9f Ready   1m   v1.5.3
```

Команда `kubectl get` может выводить список всех видов объектов Kubernetes. Вы будете использовать его постоянно, но она обычно показывает только самую элементарную информацию для перечисленных объектов.

> **СОВЕТ.** В один из узлов можно войти с помощью команды `gcloud compute ssh <имя-узла>` и узнать, что работает на узле.

Получение дополнительных сведений об объекте

Для просмотра более подробной информации об объекте можно использовать команду `kubectl describe`, которая показывает гораздо больше:

```
$ kubectl describe node gke-kubia-85f6-node-0rrx
```

Я пропускаю фактический вывод команды `describe`, потому что он довольно обширный и будет полностью нечитаемым здесь, в книге. Результат показывает статус узла, данные его ЦП и оперативной памяти, системную информацию, работающие на узле контейнеры и многое другое.

В предыдущем примере команды `kubectl describe` вы задали имя узла явным образом, но можно было также просто выполнить `kubectl describe node` без ввода имени узла, и она бы распечатала детальное описание всех узлов.

> **СОВЕТ.** Выполнение команд `describe` и `get` без указания имени объекта бывает на руку, когда имеется всего один объект данного типа, поэтому вы не тратите время на ввод или копирование/вставку имени объекта.

Прежде чем перейти к запуску вашего первого приложения в Kubernetes, раз уж мы говорим о сокращении нажатий клавиш, позвольте дать вам дополнительные советы о том, как сделать работу с `kubectl` намного проще.

2.2.3 Настройка псевдонима и автозавершение в командной строке для kubectl

Вы будете использовать `kubectl` довольно часто. Вы скоро поймете, что необходимость набирать полную команду каждый раз – это настоящая головная боль. Прежде чем продолжить, потратьте минуту, чтобы упростить свою жизнь, установив псевдоним и автозавершение по нажатии клавиши **Tab** для `kubectl`.

Создание псевдонима

На протяжении всей книги я всегда буду использовать полное имя исполняемого файла `kubectl`, но вы можете добавить короткий псевдоним, такой как `k`, поэтому вам не придется вводить `kubectl` каждый раз. Если вы еще не использовали псевдонимы, вот как их следует определять. Добавьте следующую ниже строку в файл `~/.bashrc` или эквивалентный ему файл:

```
alias k=kubectl
```

> **ПРИМЕЧАНИЕ.** Если для настройки кластера вы использовали `gcloud`, то у вас, возможно, уже есть исполняемый файл `k`.

Конфигурирование автодополнения по нажатии клавиши Tab для kubectl

Даже с коротким псевдонимом, таким как `k`, вам все равно нужно набирать на клавиатуре гораздо больше, чем вам хотелось бы. К счастью, команда `kubectl` также может выводить автодополнение для оболочек bash и zsh. Он активирует автодополнение по нажатии **Tab** не только для имен команд, но и для имен реальных объектов. Например, вместо того чтобы писать полное имя узла в предыдущем примере, вам нужно только набрать

```
$ kubectl desc<TAB> no<TAB> gke-ku<TAB>
```

Для того чтобы активировать автозавершение по нажатии **Tab** в bash, вам сначала нужно установить пакет под названием `bash-completion`, а затем выполнить следующую ниже команду (вы, вероятно, также захотите добавить его в `~/.bashrc` или его эквивалент):

```
$ source <(kubectl completion bash)
```

Правда, тут есть одно предостережение. При выполнении предыдущей команды автодополнение по нажатии **Tab** будет работать только при использовании полного имени `kubectl` (оно не будет работать при использовании

псевдонима k). Для того чтобы исправить это, необходимо немного преобразовать результат команды автодополнения kubectl:

```
$ source <(kubectl completion bash | sed s/kubectl/k/g)
```

> **ПРИМЕЧАНИЕ.** К сожалению, в момент написания этой главы автодополнение команд оболочки не работает для псевдонимов в MacOS. Для того чтобы автозавершение работало, необходимо использовать полное имя команды kubectl.

Теперь все готово для начала взаимодействия с кластером без необходимости набирать слишком много на клавиатуре. Наконец, вы можете запустить свое первое приложение на Kubernetes.

2.3 Запуск первого приложения на Kubernetes

Поскольку это может быть для вас впервые, вы будете использовать самый простой способ запуска приложений на Kubernetes. Обычно вы готовите манифест JSON или YAML, содержащий описание всех компонентов, которые вы хотите развернуть, но поскольку мы еще не говорили о типах компонентов, которые вы можете создавать в Kubernetes, вы будете использовать для запуска простую однострочную команду.

2.3.1 Развертывание приложения Node.js

Самый простой способ развертывания приложения – применить команду kubectl run, которая создаст все необходимые компоненты без необходимости работать с JSON или YAML. Благодаря этому нам не нужно погружаться в структуру каждого объекта. Попробуйте запустить образ, созданный и отправленный ранее в Docker Hub. Вот как он запускается в Kubernetes:

```
$ kubectl run kubia --image=luksa/kubia --port=8080 --generator=run/v1
replicationcontroller "kubia" created
```

Часть команды --image=luksa/kubia явным образом задает образ контейнера, который вы хотите запустить, а параметр --port=8080 сообщает Kubernetes, что ваше приложение прослушивает порт 8080. Однако последний флаг (--generator) требует объяснения. Обычно вы им не будете пользоваться, но вы используете его здесь, поэтому вместо *развертывания* (компонент Deployment) Kubernetes создает *контроллер репликации* (компонент ReplicationController). Позже в этой главе вы узнаете, что такое контроллеры репликации, а вот о развертываниях мы не будем говорить вплоть до главы 9. Вот почему я не хочу, чтобы kubectl создавал развертывание.

Как видно из результата предыдущей команды, был создан контроллер репликации с именем kubia. Как уже отмечалось, мы увидим, что это такое, позже в этой главе. На данный момент давайте начнем с нижней части и сосредо-

точимся на созданном контейнере (вы можете допустить, что контейнер был создан, потому что вы указали образ контейнера в команде `run`).

Первое знакомство с модулями

Вам, возможно, интересно, сможете ли вы увидеть свой контейнер в списке, показывающем все запущенные контейнеры. Может быть, что-то вроде `kubectl get containers`? Дело в том, что это не совсем то, как работает Kubernetes. Эта система не общается с индивидуальными контейнерами напрямую. Вместо этого она использует концепцию нескольких совместно расположенных контейнеров. Эта группа контейнеров называется модулем.

Модуль (pod) – это группа, состоящая из одного или нескольких тесно связанных контейнеров, которые всегда будут выполняться вместе на одном рабочем узле и в одном пространстве имен Linux. Каждый модуль подобен отдельной логической машине с собственным IP-адресом, сетевым именем, процессами и т. д., выполняющей одно приложение. Приложение может быть одним процессом, выполняемым в одном контейнере, или же главным процессом приложения и дополнительными вспомогательными процессами, каждый из которых выполняется в своем собственном контейнере. Все контейнеры в модуле будут работать на одной логической машине, в то время как контейнеры в других модулях, даже если они работают на одном рабочем узле, будут работать на другом.

Для того чтобы лучше понять взаимосвязь между контейнерами, модулями и узлами, изучите рис. 2.5. Как вы можете видеть, каждый модуль имеет свой собственный IP-адрес и содержит один или несколько контейнеров, каждый из которых запускает приложение.

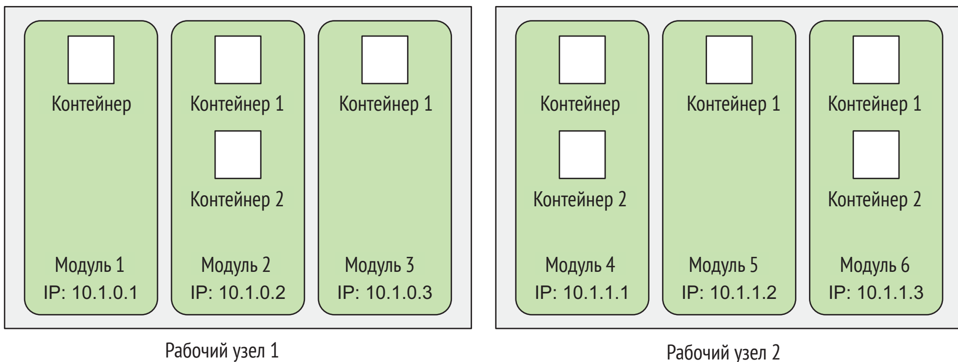


Рис. 2.5. Взаимосвязь между контейнерами, модулями и физическими рабочими узлами

Список модулей

Поскольку вы не можете вывести список отдельных контейнеров, так как они не являются автономными объектами Kubernetes, можно ли вместо этого вывести список модулей? Да, можно. Давайте посмотрим в следующем ниже листинге, как поручить `kubectl` вывести список модулей.

Листинг 2.14. Вывод списка модулей

```
$ kubectl get pods
NAME         READY  STATUS   RESTARTS  AGE
kubia-4jfyf  0/1    Pending  0         1m
```

Это ваш модуль. Он пока еще находится в статусе ожидания (`Pending`), и единственный контейнер модуля показан как еще не готовый (об этом говорят 0/1 в столбце `READY`). Причина, по которой модуль еще не работает, заключается в том, что рабочий узел, которому назначен модуль, скачивает образ контейнера перед его запуском. Когда скачивание будет завершено, контейнер модуля будет создан, а затем, как показано в следующем ниже листинге, модуль перейдет в состояние выполнения `Running`.

Листинг 2.15. Повторный вывод списка модулей, чтобы увидеть изменение статуса модуля

```
$ kubectl get pods
NAME         READY  STATUS   RESTARTS  AGE
kubia-4jfyf  1/1    Running  0         5m
```

Для того чтобы увидеть дополнительные сведения о модуле, можно также использовать команду `kubectl describe pod`, как это было ранее для одного из рабочих узлов. Если модуль застрял в статусе ожидания, возможно, Kubernetes не сможет выгрузить образ из хранилища. Если вы используете собственный образ, убедитесь, что он помечен как общедоступный в хранилище Docker Hub. Для того чтобы убедиться, что образ может быть успешно выгружен, попробуйте выгрузить образ вручную с помощью команды `docker pull` на другой машине.

Что произошло за кулисами

Для того чтобы помочь вам визуализировать то, что произошло, посмотрите на рис. 2.6. Он показывает оба шага, которые вы должны были выполнить, чтобы получить образ контейнера, работающий внутри Kubernetes. Сначала вы создали образ и переместили его в хранилище Docker Hub. Это было необходимо, поскольку создание образа на локальной машине делает его доступным только на локальной машине, но необходимо сделать его доступным для демонов Docker, работающих на рабочих узлах.

Когда вы выполнили команду `kubectl`, она создала в кластере новый объект контроллера репликации (ReplicationController), отправив запрос REST HTTP на сервер Kubernetes API. Затем контроллер репликации создал новый модуль, который планировщик запланировал для одного из рабочих узлов. Агент Kubelet на этом узле увидел, что модуль был назначен рабочему узлу, и поручил платформе Docker выгрузить указанный образ из хранилища, потому что образ не был доступен локально. После скачивания образа платформа Docker создала и запустила контейнер.

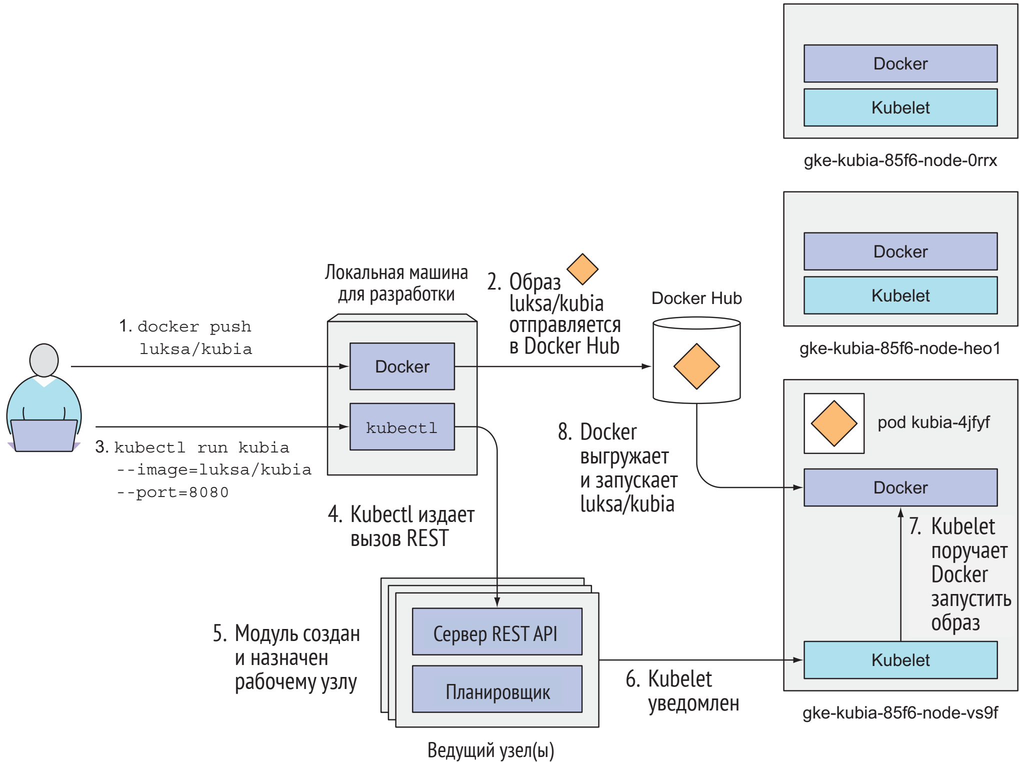


Рис. 2.6. Запуск образа контейнера luksa/kubia в Kubernetes

Остальные два узла показаны, чтобы продемонстрировать контекст. Они не играли никакой роли в этом процессе, потому что модуль не был запланирован для них.

> **ОПРЕДЕЛЕНИЕ.** Термин *планирование* означает назначение модуля узлу. Модуль запускается сразу же, а не в какое-то время в будущем, о чем, возможно, вы могли подумать при виде этого термина.

2.3.2 Доступ к веб-приложению

Ваш модуль работает, тогда как получить к нему доступ? Мы отметили, что каждый модуль получает свой собственный IP-адрес, но этот адрес является внутренним для кластера и не доступен извне. Для того чтобы сделать модуль доступным извне, вы предоставляете к нему доступ через объект службы (Service). Вы создадите специальную службу с типом `LoadBalancer`, потому что если вы создадите обычную службу (службу `ClusterIP`), то так же, как и модуль, она будет доступна только изнутри кластера. При создании службы с типом `LoadBalancer` будет создана внешняя подсистема балансировки нагрузки, и вы сможете подключаться к модулю через общедоступный IP-адрес подсистемы балансировки нагрузки.

Создание объекта Service

Для того чтобы создать службу, надо сообщить Kubernetes обеспечить доступ к контроллеру репликации, созданному ранее:

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
service "kubia-http" exposed
```

> **ПРИМЕЧАНИЕ.** Вместо `replicationcontroller` мы используем аббревиатуру `rc`. Такую аббревиатуру имеет большинство типов ресурсов, поэтому вам не нужно вводить полное имя (например, `po` для модулей `pods`, `svc` для служб `services` и т. д.).

Вывод списка служб

Результат команды `expose` упоминает службу под названием `kubia-http`. Службы – это такие же объекты, как и модули (Pod) и узлы (Node), поэтому созданный объект службы можно просмотреть, выполнив команду `kubectl get services`, как показано в следующем ниже листинге.

Листинг 2.16. Вывод списка служб

```
$ kubectl get services
NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
kubernetes   10.3.240.1     <none>        443/TCP         34m
kubia-http   10.3.246.185   <pending>     8080:31348/TCP  4s
```

В списке показаны две службы. Пока проигнорируйте службу `kubernetes` и внимательно изучите созданную службу `kubia-http`. У нее еще нет внешнего IP-адреса, так как для создания подсистемы балансировки нагрузки облачной инфраструктуре Kubernetes требуется время. Как только подсистема балансировки нагрузки будет включена, должен отобразиться внешний IP-адрес службы. Давайте подождем некоторое время и снова выведем список служб, как показано в следующем ниже листинге.

Листинг 2.17. Повторный вывод списка служб, чтобы узнать, назначен внешний IP-адрес или нет

```
$ kubectl get svc
NAME         CLUSTER-IP     EXTERNAL-IP     PORT(S)         AGE
kubernetes   10.3.240.1     <none>          443/TCP         35m
kubia-http   10.3.246.185   104.155.74.57   8080:31348/TCP  1m
```

Ага, есть внешний IP. Ваше приложение теперь доступно по адресу `http://104.155.74.57:8080` из любой точки мира.

ПРИМЕЧАНИЕ. Minikube не поддерживает службы балансировщика нагрузки `LoadBalancer`, поэтому служба никогда не получит внешний IP. Но вы можете получить доступ к службе в любом случае через его внешний порт. Как это сделать, описано в совете из следующего раздела.

Доступ к вашей службе через внешний IP-адрес

Теперь можно отправлять запросы в модуль через внешний IP-адрес и порт службы:

```
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Класс! Ваше приложение теперь работает где-то в вашем трехузловом кластере Kubernetes (или в одноузловом кластере, если вы используете Minikube). Если вы не ведете учет шагов, необходимых для настройки всего кластера, то, для того чтобы запустить приложение и сделать его доступным для пользователей по всему миру, достаточно выполнить всего две простые команды.

СОВЕТ. При использовании Minikube можно получить IP-адрес и порт, через который можно обращаться к службе, выполнив команду `minikube service kubia-http`.

Если вы посмотрите внимательно, то увидите, что приложение сообщает имя модуля как хостнейм. Как уже упоминалось, каждый модуль ведет себя как отдельная независимая машина с собственным IP-адресом и хостнеймом. Несмотря на то что это приложение работает в операционной системе рабочего узла, оно выглядит так, как будто оно работает на отдельной машине, выделенной для самого приложения, – никакие другие процессы не выполняются вместе с ним.

2.3.3 Логические части вашей системы

До сих пор я в основном объяснял реальные физические компоненты вашей системы. Вы имеете три рабочих узла, то есть виртуальные машины, выполняющие платформу Docker и агента Kubelet, и у вас есть ведущий узел, который контролирует всю систему. Честно говоря, мы не знаем, размещены ли все индивидуальные компоненты плоскости управления Kubernetes на единственном ведущем узле или они разделены по нескольким узлам. На самом деле это не имеет значения, потому что вы взаимодействуете только с сервером API, который доступен в одной конечной точке.

Помимо данного физического представления системы, есть также отдельный, логический взгляд на нее. Я уже упоминал такие объекты, как модули (Pod), контроллеры репликации (ReplicationController) и службы (Service). Все они будут объяснены в следующих нескольких главах, но давайте быстро посмотрим, как они сочетаются друг с другом и какие роли они играют в вашей маленькой конфигурации.

Как согласуются друг с другом контроллер репликации, модуль и служба

Как я уже объяснил, вы не создаете и не работаете с контейнерами напрямую. Вместо этого основным строительным блоком в Kubernetes является модуль. Но вы, по правде говоря, и не создавали никаких модулей, по крайней мере не напрямую. Выполнив команду `kubectl run`, вы создали контроллер репликации, и именно этот контроллер репликации создал фактический объект-модуль Pod. Для того чтобы сделать этот модуль доступным за пределами кластера, вы поручили Kubernetes предоставить доступ ко всем модулям, управляемым этим контроллером репликации, как к единой службе. Примерная картина всех трех элементов представлена на рис. 2.7.

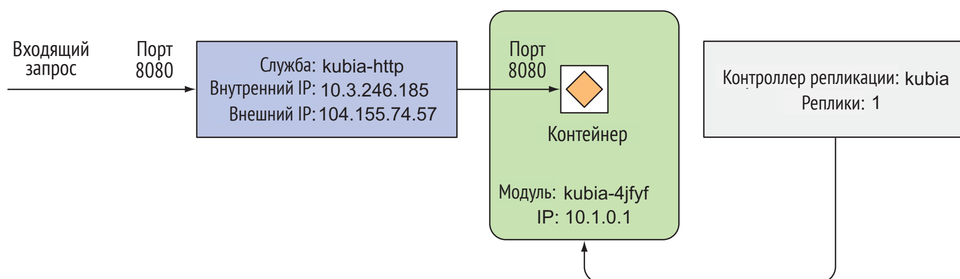


Рис. 2.7. Ваша система состоит из контроллера репликации, модуля и службы

Модуль и его контейнер

Главным и наиболее важным компонентом в вашей системе является модуль. Он содержит всего один контейнер, но обычно модуль может содержать столько контейнеров, сколько вы захотите. Внутри контейнера находится ваш процесс Node.js, который привязан к порту 8080 и ожидает HTTP-запросов. Модуль имеет свой собственный уникальный приватный IP-адрес и хостнейм.

Роль контроллера репликации

Следующим компонентом является контроллер репликации `kubia`. Он гарантирует, что всегда имеется ровно один экземпляр вашего модуля. В общем случае контроллер репликации используется для репликации модулей (то есть создания нескольких копий модуля) и поддержания их работы. В вашем случае вы не указали, сколько реплик модуля требуется, поэтому контроллер репликации создал единственный. Если ваш модуль по какой-то причине исчезнет, то контроллер репликации создаст новый модуль для замены отсутствующего.

Зачем нужна служба

Третьим компонентом системы является служба `kubia-http`. Для того чтобы понять, почему вам нужны службы, вам нужно понять ключевую особенность модулей. Они эфемерны. Модуль может исчезнуть в любое время – потому что узел, на котором он запущен, аварийно завершил работу, или потому

что кто-то уничтожил модуль, или потому что модуль был вытеснен, чтобы обеспечить здоровое состояние узла. При любом из этих событий контроллер репликации заменяет отсутствующий модуль новым модулем, как описано выше. Этот новый модуль получает IP-адрес, отличающийся от того, который был у заменяемого модуля. Именно здесь вступают в игру службы – чтобы разрешать проблему изменчивых IP-адресов модуля, а также обеспечивать доступ ко множеству модулей по единой постоянной паре IP-адреса и порта.

Когда служба создается, она получает статический IP-адрес, который не меняется в течение срока действия службы. Вместо прямого подключения к модулям клиенты должны подключаться к службе через ее постоянный IP-адрес. Служба удостоверяется, что один из модулей получает подключение, независимо от того, где модуль в настоящее время работает (и каким является его IP-адрес).

Службы представляют собой статическое расположение для группы из одного или нескольких модулей, которые предоставляют одну и ту же службу. Запросы, поступающие на IP-адрес и порт службы, будут перенаправлены на IP-адрес и порт одного из модулей, принадлежащих службе на данный момент.

2.3.4 Горизонтальное масштабирование приложения

Теперь у вас есть работающее приложение, которое отслеживается и поддерживается в рабочем состоянии контроллером репликации и доступ к которому обеспечивается внешнему миру через службу. Теперь давайте совершим дополнительное волшебство.

Одним из главных преимуществ использования Kubernetes является простота масштабирования приложений. Давайте посмотрим, как легко масштабируется количество модулей. Вы увеличите количество запущенных экземпляров до трех.

Ваш модуль управляется контроллером репликации. Давайте посмотрим на это с помощью команды `kubectl get`:

```
$ kubectl get replicationcontrollers
NAME    DESIRED   CURRENT   AGE
kubia   1         1         17m
```

Вывод списка всех типов ресурсов с помощью команды kubectl get

Вы использовали одну и ту же базовую команду `kubectl get` для вывода списка объектов в кластере. Эта команда используется для вывода списка объектов: узлов (Node), модулей (Pod), служб (Service) и контроллеров репликации (ReplicationController). Список всех возможных типов объектов можно получить, вызвав `kubectl get` без указания типа. Затем эти типы можно использовать с различными командами `kubectl`, такими как `get`, `describe` и т. д. В списке также указаны сокращения, о которых я упоминал ранее.

В списке показан единственный контроллер репликации под названием kubia. В столбце DESIRED показано количество реплик модуля, которые должен хранить контроллер репликации, а в столбце CURRENT – фактическое количество запущенных модулей. В вашем случае вы хотели бы иметь одну реплику работающего модуля, и в настоящее время выполняется ровно одна реплика.

Увеличение количества требуемых реплик

Для того чтобы отмасштабировать количество реплик вашего модуля, необходимо изменить требуемое количество реплик на контроллере репликации. Это делается следующим образом:

```
$ kubectl scale rc kubia --replicas=3
replicationcontroller "kubia" scaled
```

Теперь вы поручили Kubernetes убедиться, что три экземпляра вашего модуля всегда работают. Обратите внимание, что вы не указали Kubernetes, какие действия предпринять. Вы не сообщили ему добавить еще два модуля. Вы только устанавливаете новое требуемое количество экземпляров и позволяете системе Kubernetes определить, какие действия она должна предпринять, чтобы достигнуть запрошенного состояния.

Это один из самых фундаментальных принципов Kubernetes. Вместо того чтобы говорить системе Kubernetes точно, какие действия она должна выполнять, вы лишь декларативно изменяете требуемое состояние системы и позволяете Kubernetes исследовать текущее фактическое состояние и согласовать его с требуемым состоянием. Это справедливо для всех Kubernetes.

Просмотр результатов масштабирования вширь

Вернемся к увеличению количества реплик. Давайте снова выведем список контроллеров репликации, чтобы увидеть обновленное количество реплик:

```
$ kubectl get rc
NAME    DESIRED   CURRENT   READY   AGE
kubia   3         3         2       17m
```

Поскольку фактическое количество модулей уже увеличено до трех (как видно из столбца CURRENT), вывод списка всех модулей теперь должен отображать три модуля вместо одного:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-hczji   1/1     Running   0          7s
kubia-iq9y6   0/1     Pending   0          7s
kubia-4jfyf   1/1     Running   0          18m
```

Как вы можете видеть, вместо одного теперь три модуля. Два уже запущены, один все еще находится в ожидании, но должен быть готов через несколько минут, как только образ контейнера будет скачан и контейнер запущен.

Как вы можете видеть, процедура масштабирования приложения невероятно проста. Как только приложение запущено в рабочем окружении и возникает необходимость в масштабировании приложения, дополнительные экземпляры можно добавить с помощью одной-единственной команды без необходимости установки и запуска дополнительных копий вручную.

Имейте в виду, что само по себе приложение должно поддерживать масштабирование по горизонтали. Система Kubernetes волшебным образом не делает ваше приложение масштабируемым; она только делает задачу масштабирования приложения вверх или вниз тривиальной.

Наблюдение за тем, как при поступлении в службу запросы попадают во все три модуля

Поскольку теперь у вас запущено несколько экземпляров приложения, давайте посмотрим, что произойдет, если вы снова попадете в URL-адрес службы. Будете ли вы всегда попадать в один и тот же экземпляр приложения или нет?

```
$ curl 104.155.74.57:8080
You've hit kubia-hczji
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Запросы случайным образом попадают в разные модули. Это то, что делают службы в Kubernetes, когда они поддерживаются более чем одним экземпляром модуля. Они действуют как балансировщик нагрузки, находясь в условиях множества модулей. Когда есть всего один модуль, службы предоставляют статический адрес для одного модуля. Независимо от того, поддерживается ли служба одним модулем или группой модулей, эти модули приходят и уходят при перемещении по кластеру, что означает изменение их IP-адресов, но служба всегда находится по одному адресу. Это позволяет клиентам легко подключаться к модулям независимо от того, сколько их существует и как часто они меняют местоположение.

Визуализация нового состояния системы

Давайте снова визуализируем вашу систему, чтобы увидеть, что изменилось с прошлого момента. На рис. 2.8 показано новое состояние системы. У вас по-прежнему одна служба и один контроллер репликации, зато теперь у вас три экземпляра модуля, все управляемые контроллером репликации. Служба больше не отправляет все запросы в один модуль, а распределяет их по всем трем модулям, как показано в эксперименте с `curl` в предыдущем разделе.

В качестве упражнения можно попробовать развернуть дополнительные экземпляры, еще больше увеличив число реплик контроллера репликации, а затем отмасштабировав назад.

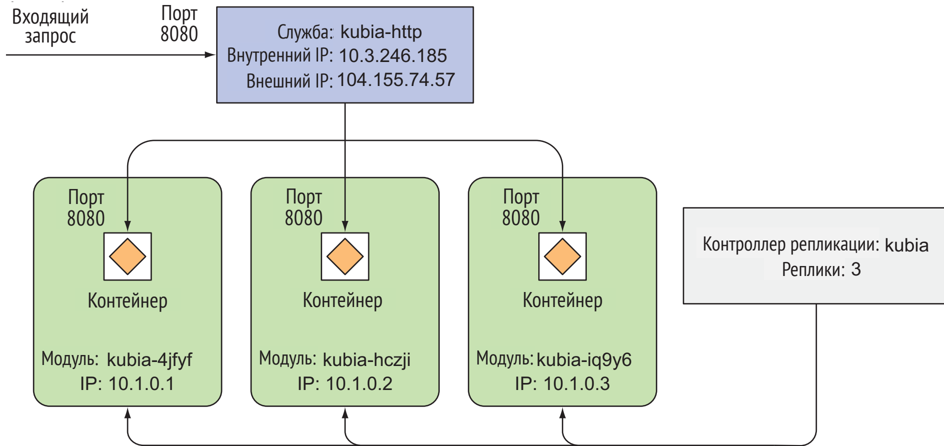


Рис. 2.8. Три экземпляра модуля, которые управляются одним контроллером репликации и доступ к которым обеспечивается через единый IP-адрес и порт службы

2.3.5 Исследование узлов, на которых запущено приложение

Возможно, вас интересует, на какие узлы ваши модули были запланированы. В мире Kubernetes ответ на вопрос, на каком узле работает модуль, не так важен, поскольку он назначается узлу, который может обеспечить нужные модулю процессор и память, чтобы работать должным образом.

Независимо от узла, к которому они приписаны, все приложения, запущенные в контейнерах, имеют одинаковый тип операционной среды. Каждый блок имеет свой собственный IP и может общаться с любым другим модулем, независимо от того, работает ли другой модуль на том же самом узле или на другом. Каждый модуль обеспечивается запрашиваемым объемом вычислительных ресурсов, поэтому не имеет никакой разницы, обеспечиваются ли эти ресурсы тем или другим узлом.

Отображение IP-адреса и узла модуля при выводе списка модулей

Если вы отнеслись внимательно, то вы, вероятно, заметили, что команда `kubectl get pods` даже не показывает никакой информации об узлах, на которые запланированы работать модули. Это вызвано тем, что обычно это не является важной частью информации.

Однако с помощью параметра `-o wide` можно запросить дополнительные столбцы для показа. При выводе списка модулей этот параметр показывает IP-адрес модуля и узел, на котором работает модуль:

```
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP         NODE
kubia-hczji   1/1     Running   0          7s    10.1.0.2   gke-kubia-85...
```

Осмотр других деталей модуля с помощью kubectl describe

Как показано в следующем ниже листинге, также можно увидеть узел с помощью команды `kubectl describe`, которая показывает много других деталей модуля.

Листинг 2.18. Описание модуля с помощью команды `kubectl describe`

```
$ kubectl describe pod kubia-hczji
Name:          kubia-hczji
Namespace:     default
Node:          gke-kubia-85f6-node-vs9f/10.132.0.3     <-- Вот узел, на котором запланирована
Start Time:    Fri, 29 Apr 2016 14:12:33 +0200             работа модуля
Labels:        run=kubia
Status:        Running
IP:            10.1.0.2
Controllers:   ReplicationController/kubia
Containers:    ...
Conditions:
  Type         Status
  Ready        True
Volumes: ...
Events: ...
```

Эта команда показывает, среди прочего, узел, к которому был приписан модуль, время, когда он был запущен, образ(ы), который(е) он выполняет, и другую полезную информацию.

2.3.6 Знакомство с панелью управления Kubernetes

Прежде чем мы завершим эту начальную практическую главу, давайте рассмотрим другой способ изучения вашего кластера Kubernetes.

До сих пор вы использовали только программу командной строки `kubectl`. Если вам больше нравятся графические веб-интерфейсы пользователя, то вы будете рады услышать, что Kubernetes также поставляется с хорошей (но все еще развивающейся) веб-панелью управления.

Панель управления позволяет просматривать список всех модулей, контроллеров репликации, служб и других объектов, развернутых в кластере, а также создавать их, изменять и удалять. На рис. 2.9 показана панель управления.

Хотя панель управления в этой книге не используется, ее можно открыть в любое время, чтобы быстро просмотреть графическое представление развернутых в кластере объектов после создания или изменения объектов с помощью `kubectl`.

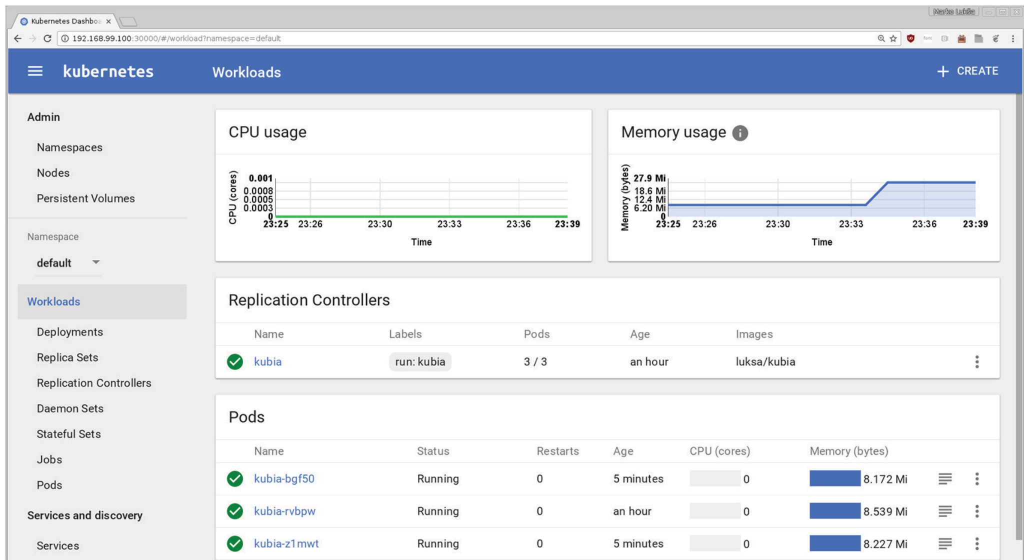


Рис. 2.9. Снимок экрана веб-панели управления Kubernetes

Доступ к панели управления при запуске Kubernetes в GKE

Если вы используете Google Kubernetes Engine, то можете узнать URL-адрес панели управления с помощью команды `kubectl cluster-info`, с которой мы уже познакомились:

```
$ kubectl cluster-info | grep dashboard
kubernetes-dashboard is running at https://104.155.108.191/api/v1/proxy/
➥  namespaces/kube-system/services/kubernetes-dashboard
```

Если открыть этот URL-адрес в браузере, то вам будет предложено ввести имя и пароль пользователя. Вы найдете имя и пароль пользователя, выполнив следующую ниже команду:

```
$ gcloud container clusters describe kubia | grep -E "(username|password):"
password: 32nENgreEJ632A12
username: admin
```

Имя и пароль пользователя для панели управления

Доступ к панели мониторинга при использовании Minikube

При использовании Minikube для управления кластером Kubernetes, чтобы открыть панель управления в браузере, выполните следующую ниже команду:

```
$ minikube dashboard
```

Панель управления откроется в браузере по умолчанию. В отличие от GKE, для доступа к ней вам не нужно будет вводить учетные данные.

2.4 Резюме

Надеюсь, эта начальная практическая глава показала вам, что Kubernetes не является сложной в использовании платформой, и вы готовы подробно узнать обо всех аспектах, которые она может предоставить. После прочтения этой главы вы должны теперь знать, как:

- выгружать и запускать любой общедоступный образ контейнера;
- упаковывать свои приложения в образы контейнеров и делать их доступными для всех, отправляя образы в хранилище удаленных образов;
- входить в работающий контейнер и проверять его среду;
- настраивать многоузловой кластер Kubernetes на Google Kubernetes Engine;
- настраивать псевдоним и автозаполнение в инструменте командной строки `kubectl`;
- выводить список и проверять узлы, модули, службы и контроллеры репликации в кластере Kubernetes;
- запускать контейнер в Kubernetes и делать его доступным за пределами кластера;
- связаны между собой модули, контроллеры репликации и службы;
- масштабировать приложение по горизонтали путем изменения количества реплик в контроллере репликации;
- получать доступ к панели управления Kubernetes на Minikube и GKE.

Глава 3

Модули: запуск контейнеров в Kubernetes

Эта глава посвящена:

- созданию, запуску и остановке модулей;
- организации модулей и других ресурсов с помощью меток;
- выполнению операции на всех модулях с определенной меткой;
- использованию пространств имен для разделения модулей на непересекающиеся группы;
- назначению модулей определенным типам рабочих узлов.

Предыдущая глава должна была дать вам примерную картину основных компонентов, которые вы создаете в Kubernetes, и, по крайней мере, схематичный план того, что они делают. Теперь мы начнем более подробно рассматривать все типы объектов (или ресурсов) Kubernetes, чтобы вы понимали, когда, как и зачем использовать каждый из них. Мы начнем с модулей, потому что они являются центральной, самой важной концепцией в Kubernetes. Все остальное либо управляет модулями, обеспечивает к ним доступ, либо используется модулями.

3.1 Знакомство с модулями

Вы уже узнали, что модуль – это размещенная рядом группа контейнеров, которая представляет собой основной строительный блок в Kubernetes. Вместо развертывания контейнеров по отдельности вы всегда развертываете и оперируете модулем контейнеров. Мы не утверждаем, что модуль всегда включает более одного контейнера, – обычно модули содержат всего один контейнер. Ключевой аспект модулей состоит в том, что когда модуль действительно со-

держит множество контейнеров, все они всегда выполняются на одном рабочем узле, как показано на рис. 3.1, он никогда не охватывает несколько рабочих узлов.

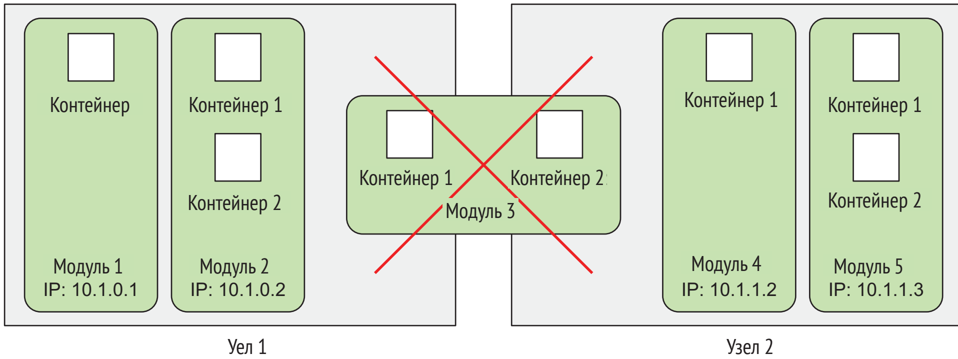


Рис. 3.1. Все контейнеры модуля работают на одном узле. Модуль никогда не охватывает двух узлов

3.1.1 Зачем нужны модули

Но зачем нам вообще нужны модули? Почему нельзя использовать контейнеры напрямую? Зачем нам вообще запускать несколько контейнеров вместе? Разве мы не можем поместить все наши процессы в один контейнер? Сейчас мы ответим на эти вопросы.

Почему несколько контейнеров лучше, чем один контейнер, выполняющий несколько процессов

Представьте себе приложение, состоящее из нескольких процессов, которые обмениваются либо через IPC (Inter-Process Communication, межпроцессное взаимодействие), либо через локально хранящиеся файлы, что требует их запуска на одной машине. Поскольку в Kubernetes вы всегда запускаете процессы в контейнерах, и каждый контейнер похож на изолированную машину, вы можете подумать, что имеет смысл запускать несколько процессов в одном контейнере, однако вы не должны этого делать.

Контейнеры предназначены для выполнения только одного процесса в расчете на контейнер (если сам процесс не порождает дочерние процессы). Если вы запускаете несколько несвязанных процессов в одном контейнере, вы несете ответственность за то, чтобы все эти процессы работали, управляли своими журналами и т. д. Например, вам пришлось бы включить механизм автоматического перезапуска отдельных процессов в случае их сбоя. Кроме того, все эти процессы будут делать вывод в одном и том же стандартном выводе, поэтому вам будет трудно понять, какой процесс что вывел.

Поэтому необходимо запускать каждый процесс в собственном контейнере. Именно так Docker и Kubernetes задуманы использоваться.

3.1.2 Общее представление о модулях

Поскольку вы не должны группировать несколько процессов в один контейнер, очевидно, что вам нужна другая конструкция более высокого уровня, которая позволит вам связывать контейнеры вместе и управлять ими как единым целым. Это рассуждение лежит в основе модулей.

Модуль, состоящий из контейнеров, позволяет вам выполнять родственные процессы совместно и обеспечивать их (почти) одинаковой средой, как если бы все они работали в едином контейнере, при этом держа их несколько изолированно. Благодаря этому вы получите лучшее из обоих миров. Вы сможете воспользоваться всеми возможностями, которые предоставляют контейнеры, и в то же время давать процессам иллюзию совместной работы.

Частичная изоляция между контейнерами одного модуля

В предыдущей главе вы узнали, что контейнеры полностью изолированы друг от друга, но теперь вы видите, что вместо отдельных контейнеров вам требуется изолировать группы контейнеров. Вы хотите, чтобы контейнеры внутри каждой группы совместно использовали определенные ресурсы, хотя и не все, чтобы они не были полностью изолированы. Kubernetes достигает этого путем настройки Docker так, что, вместо того чтобы каждый контейнер имел свой собственный набор пространств имен Linux, все контейнеры модуля совместно используют один и тот же набор.

Поскольку все контейнеры модуля работают в тех же самых пространствах имен Network и UTS (здесь речь идет о пространствах имен Linux), все они используют один хостнейм и сетевые интерфейсы. Кроме того, все контейнеры модуля работают под одинаковым пространством имен IPC и могут общаться через IPC. В последних версиях Kubernetes и Docker они также могут совместно использовать одно пространство имен PID, но этот функционал не включен по умолчанию.

> **ПРИМЕЧАНИЕ.** Когда контейнеры одного и того же модуля используют раздельные пространства имен PID, при запуске ps aux в контейнере вы увидите только собственные процессы контейнера.

Но когда дело касается файловой системы, все немного по-другому. Поскольку подавляющая часть файловой системы контейнера поступает из образа контейнера, по умолчанию файловая система каждого контейнера полностью изолирована от других контейнеров. Однако вполне возможно позволить им совместно использовать их файловые каталоги, применяя концепцию Kubernetes под названием *том* (Volume), о которой мы поговорим в главе 6.

Как контейнеры используют одно и то же пространство IP-адресов и портов

Здесь следует подчеркнуть один момент, который состоит в том, что, поскольку контейнеры в модуле работают в одинаковом пространстве имен

Network, они совместно используют одинаковое пространство IP-адресов и портов. Это означает, что процессы, запущенные в контейнерах одного модуля, должны заботиться о том, чтобы не привязываться к тем же номерам портов, иначе они столкнутся с конфликтами портов. Но это касается только контейнеров в том же модуле. Контейнеры разных модулей никогда не смогут столкнуться с конфликтом портов, так как каждый модуль имеет отдельное пространство портов. Все контейнеры в модуле также имеют один и тот же loopback-интерфейс, поэтому контейнер может взаимодействовать с другими контейнерами в том же модуле через localhost.

Знакомство с плоской межмодульной сетью

Все модули в кластере Kubernetes находятся в одном плоском общем пространстве сетевых адресов (показано на рис. 3.2), имея в виду, что каждый модуль может получать доступ к каждому другому модулю по IP-адресу другого модуля. Между ними нет шлюзов NAT (трансляции сетевых адресов). Когда два модуля отправляют сетевые пакеты друг другу, каждый из них видит фактический IP-адрес другого в качестве IP-адреса отправителя в пакете.

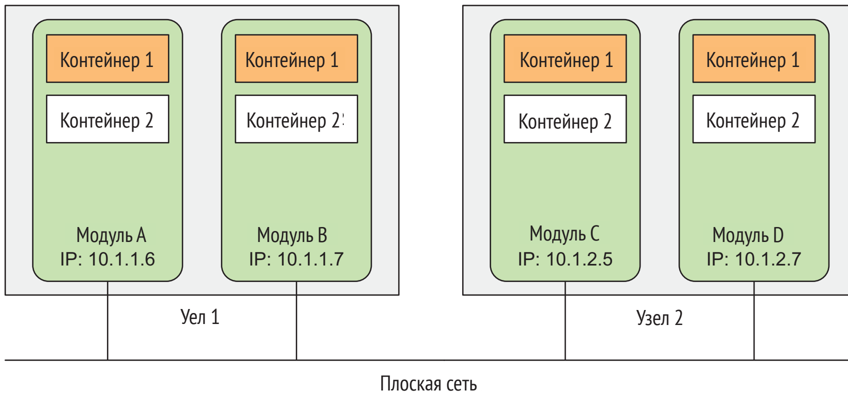


Рис. 3.2. Каждый модуль получает маршрутизируемый IP-адрес, и все другие модули видят модуль под этим IP-адресом

Следовательно, обмен между модулями всегда прост. Не имеет значения, запланированы ли два модуля для работы на одном или на разных рабочих узлах; в обоих случаях контейнеры внутри этих модулей могут взаимодействовать друг с другом через плоскую сеть без NAT, подобно компьютерам в локальной сети (LAN), независимо от фактической топологии межузловой сети. Как и компьютер в локальной сети, каждый модуль получает свой собственный IP-адрес и доступен из всех других модулей через эту сеть, сформированную специально для модулей. Обычно это достигается с помощью дополнительной программно определяемой сети, уложенной поверх реальной сети.

Подводя итог тому, что было рассмотрено в данном разделе, скажем, что модули являются логическими хостами и ведут себя как физические хосты

или виртуальные машины в неконтейнерном мире. Процессы, запущенные в одном модуле, подобны процессам, запущенным на одной физической или виртуальной машине, за исключением того, что каждый процесс инкапсулируется в контейнер.

3.1.3 Правильная организация контейнеров между модулями

Модули нужно представлять как отдельные машины, но каждая из которых размещает лишь определенное приложение. В отличие от прошлых времен, когда мы обычно втискивали разного рода приложения в один и тот же хост, мы это не делаем с модулями. Поскольку модули относительно облегчены, вы можете иметь их столько, сколько вам нужно, не производя почти никаких накладных расходов. Вместо того чтобы запихивать все в один модуль, вам следует организовывать приложения в несколько модулей, каждый из которых содержит только тесно связанные компоненты или процессы.

В связи с этим как, по вашему мнению, следует сконфигурировать многоуровневое приложение, состоящее из фронтенда с сервером приложений и бэкенда с базой данных: как один модуль или как два модуля?

Разбиение многоуровневых приложений на несколько модулей

Хотя ничто не мешает вам работать как с фротенд-сервером, так и с бэкенд-базой данных в одном модуле с двумя контейнерами, это не самый подходящий способ. Мы сказали, что все контейнеры одного и того же модуля всегда работают совместно, но разве веб-сервер и база данных обязательно должны работать на одной и той же машине? Совершенно очевидно, что нет, поэтому вы не будете помещать их в один модуль. Но разве неправильно, несмотря ни на что, все равно сделать так? В некотором смысле да.

Если и фронтенд-, и бэкенд-серверы находятся в одном модуле, то оба всегда будут работать на одной машине. Если у вас кластер Kubernetes с двумя узлами и только один модуль, то вы будете использовать только один рабочий узел и не будете использовать вычислительные ресурсы (ЦП и память), имеющиеся в вашем распоряжении на втором узле. Разбиение модуля на два позволит Kubernetes запланировать фронтенд на один узел и бэкенд-сервер на другой узел, тем самым улучшив использование вашей инфраструктуры.

Разделение на несколько модулей для активации индивидуального масштабирования

Еще одна причина, почему вам не следует помещать их в один модуль, – это масштабирование. Модуль является также базовой единицей масштабирования. Kubernetes не может горизонтально масштабировать отдельные контейнеры; вместо этого он масштабирует модули целиком. Если ваш модуль состоит из фронтенд- и бэкенд-контейнеров, когда вы масштабируете количество экземпляров модуля, скажем, до двух, вы в конечном счете окажетесь с двумя фронтенд- и двумя бэкенд-контейнерами.

Обычно фронтенд-компоненты имеют совершенно иные потребности к масштабированию, чем бэкенд, поэтому мы склонны масштабировать их по отдельности. Не говоря уже о том, что бэкенды, такие как базы данных, как правило, намного сложнее масштабировать, по сравнению с фронтенд-веб-серверами (которые не отслеживают состояния). Если возникает потребность масштабировать контейнер отдельно, то это явный признак того, что его необходимо развернуть в отдельном модуле.

Когда использовать несколько контейнеров в модуле

Основная причина размещения нескольких контейнеров в одном модуле возникает, когда приложение состоит из одного главного процесса и одного или нескольких дополнительных процессов, как показано на рис. 3.3.

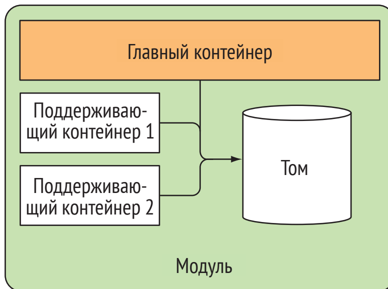


Рис. 3.3. Модули должны содержать тесно связанные контейнеры, обычно главный контейнер и контейнеры, которые поддерживают главный

Например, главным контейнером в модуле может быть веб-сервер, обслуживающий файлы из определенного файлового каталога, а дополнительный контейнер (побочный контейнер) периодически скачивает содержимое из внешнего источника и сохраняет его в каталоге веб-сервера. В главе 6 вы увидите, что вам нужно использовать том (Volume) Kubernetes, который вы монтируете в оба контейнера.

Другие примеры побочных контейнеров включают ротаторы и сборщики лог-файлов, обработчики данных, коммуникационные адаптеры и другие.

Принятие решения об использовании нескольких контейнеров в модуле

Подводя итоги тому, как контейнеры должны быть сгруппированы в модули – во время принятия решения о том, поместить ли два контейнера в один модуль или в два отдельных модуля, вам всегда нужно задавать себе следующие вопросы:

- они должны работать вместе или они могут работать на разных хостах?
- представляют ли они собой единое целое или являются независимыми компонентами?
- они должны масштабироваться вместе или по отдельности?

В принципе, вы всегда должны тяготеть к запуску контейнеров в отдельных модулях, если только конкретная причина не требует, чтобы они были частью одного и того же модуля. Рисунок 3.4 поможет вам запомнить это.

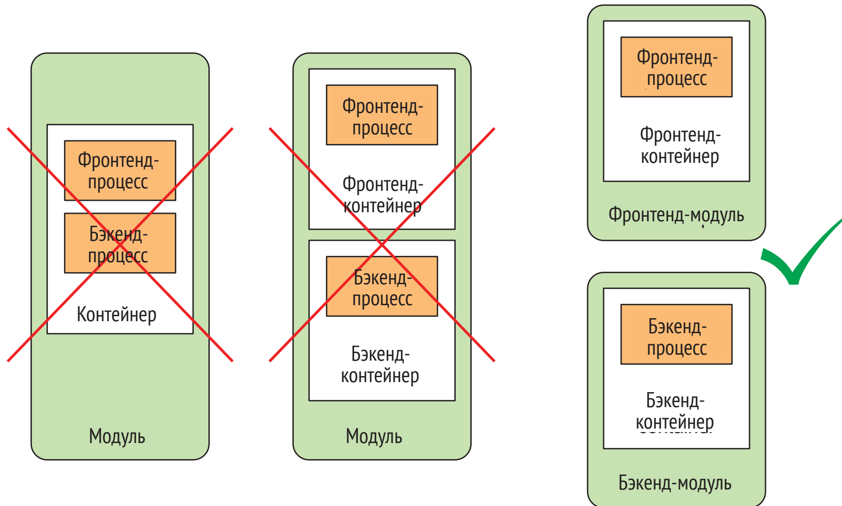


Рис. 3.4. Контейнер не должен выполнять несколько процессов. Модуль не должен содержать многочисленные контейнеры, если они не обязательно должны выполняться на одинаковой машине

Хотя модули могут содержать множество контейнеров, в этой главе, чтобы пока не усложнять, вы будете иметь дело только с одноконтейнерными модулями. Вы увидите, каким образом использовать несколько контейнеров в одном модуле, позже, в главе 6.

3.2 Создание модулей из дескрипторов YAML или JSON

Модули и другие ресурсы Kubernetes обычно создаются путем публикации манифеста JSON или YAML в конечной точке API REST Kubernetes. Кроме того, можно использовать другие, более простые способы создания ресурсов, такие как команда `kubectl run`, которую вы использовали в предыдущей главе, но они обычно позволяют настраивать только ограниченный набор свойств, далеко не все. Кроме того, определение всех объектов Kubernetes из файлов YAML позволяет хранить их в системе управления версиями со всеми преимуществами, которые она приносит.

Для настройки всех аспектов каждого типа ресурсов необходимо знать и разбираться в том, как определяются объекты в API Kubernetes. Вы познакомитесь с большинством из них, когда узнаете о каждом типе ресурсов по ходу чтения этой книги. Мы не будем объяснять каждое свойство, поэтому при создании объектов вы также должны обратиться к справочной документации по API Kubernetes по адресу http://kubernetes.io/docs/reference/.

3.2.1 Исследование дескриптора YAML существующего модуля

У вас уже есть некоторые существующие модули, которые вы создали в предыдущей главе, поэтому давайте посмотрим, как выглядит определение YAML для одного из этих модулей. Вы будете использовать команду `kubectl get` с параметром `-o yaml` для получения полного определения YAML модуля, как показано в следующем ниже листинге.

Листинг 3.1. Полный YAML развернутого модуля

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: ...
  creationTimestamp: 2016-03-18T12:37:50Z
  generateName: kubia
  labels:
    run: kubia
  name: kubia-zxzij
  namespace: default
  resourceVersion: «294»
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004
spec:
  containers:
  - image: luksa/kubia
    imagePullPolicy: IfNotPresent
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
    resources:
      requests:
        cpu: 100m
    terminationMessagePath: /dev/termination-log
    volumeMounts:
    - mountPath: /var/run/secrets/k8s.io/servacc
      name: default-token-kvcqa
      readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: gke-kubia-e8fe08b8-node-txje
  restartPolicy: Always
  serviceAccount: default
```

Версия API Kubernetes, используемая в этом дескрипторе YAML

Тип объекта/ресурса Kubernetes

Метаданные модуля (имя, метки, аннотации и т. д.)

Спецификация/содержание модуля (список контейнеров модуля, его томов и т. д.)

```
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  volumes:
  - name: default-token-kvcqa
    secret:
      secretName: default-token-kvcqa
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: null
    status: "True"
    type: Ready
  containerStatuses:
  - containerID: docker://f0276994322d247ba...
    image: luksa/kubia
    imageID: docker://4c325bcc6b40c110226b89fe...
    lastState: {}
    name: kubia
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2016-03-18T12:46:05Z
  hostIP: 10.132.0.4
  phase: Running
  podIP: 10.0.2.3
  startTime: 2016-03-18T12:44:32Z
```


Спецификация/ содержание модуля (список контейнеров модуля, его томов и т. д.)

Подробный статус модуля и его контейнеров

Я знаю, что это содержимое выглядит сложным, но оно становится простым, как только вы разберетесь в основах и узнаете, как отличать важные части от мелких деталей. Кроме того, вы можете успокоить себя тем, что во время создания нового модуля текст YAML, который вам нужно будет писать, будет намного короче, как вы убедитесь позже.

Знакомство с главными частями определения модуля

Определение модуля состоит из нескольких частей. Во-первых, это версия API Kubernetes, используемая в YAML, и тип ресурса, который описывает YAML. Затем почти во всех ресурсах Kubernetes находятся три важные секции:

- *метаданные* (metadata) – включают имя, пространство имен, метки и другую информацию о модуле;
- *спецификация* (spec) – содержит фактическое описание содержимого модуля, например контейнеры модуля, тома и другие данные;
- *статус* (status) – содержит текущую информацию о работающем модуле, такую как условие, в котором находится модуль, описание и статус каждого контейнера, внутренний IP модуля, и другую базовую информацию.

В листинге 3.1 показано полное описание работающего модуля, включая его статус. Секция `status` содержит данные времени выполнения только для чтения, которые показывают состояние ресурса в текущий момент. При создании нового модуля никогда не требуется указывать секцию `status`.

Три секции, описанные ранее, показывают типичную структуру объекта API Kubernetes. Как вы будете убеждаться на протяжении всей книги, все другие объекты имеют ту же анатомию. Это относительно облегчает понимание новых объектов.

Обсуждение всех индивидуальных свойств в приведенном выше YAML не имеет особого смысла, поэтому давайте посмотрим, как выглядит самый базовый YAML для создания модуля.

3.2.2 Создание простого дескриптора YAML для модуля

Вам следует создать файл под названием `kubia-manual.yaml` (вы можете создать его в любом каталоге, в котором захотите) либо скачать архив кода, прилагаемого к этой книге, где вы найдете архив внутри папки *Chapter03*. В следующем ниже листинге показано все содержимое файла.

Листинг 3.2. Базовый манифест модуля: kubia-manual.yaml

```
apiVersion: v1                      <-- Описание соответствует версии v1 API Kubernetes
kind: Pod                           <-- Вы описываете модуль
metadata:
  name: kubia-manual                <-- Имя модуля
spec:
  containers:
  - image: luksa/kubia              <-- Образ контейнера, из которого создать контейнер
    name: kubia                     <-- Имя контейнера
    ports:
    - containerPort: 8080           <-- Порт, на котором приложение слушает
      protocol: TCP
```

Уверен, вы согласитесь, что это намного проще, чем определение в листинге 3.1. Рассмотрим это описание подробнее. Оно соответствует версии v1 API Kubernetes. Тип ресурса, который вы описываете, – это модуль с именем `kubia-manual`. Модуль состоит из одного контейнера на основе образа `luksa/kubia`. Вы также дали имя контейнеру и указали, что он прослушивает порт `8080`.

Указание портов контейнера

Указание портов в определении модуля является чисто информационным. Их пропуск не влияет на возможность подключения клиентов к модулю через порт. Если контейнер принимает подключения через порт, привязанный к адресу 0.0.0.0, другие модули всегда смогут подключаться к нему, даже если

порт не указан в спецификации модуля явно. Но имеет смысл явно определять порты, чтобы каждый, кто использует ваш кластер, мог быстро увидеть, к каким портам обеспечивается доступ каждым модулем. Явное определение портов также позволяет назначить каждому порту имя, что может пригодиться, как вы увидите позже в книге.

Использование команды kubectl explain, чтобы обнаружить возможные поля объектов API

При подготовке манифеста можно обратиться к справочной документации Kubernetes по адресу http://kubernetes.io/docs/api, с тем чтобы узнать, какие атрибуты поддерживаются каждым объектом API, либо можно воспользоваться командой `kubectl explain`.

Например, при создании манифеста модуля с нуля можно начать с запроса `kubectl explain pods`:

```
$ kubectl explain pods
DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource
             is created by clients and scheduled onto hosts.
FIELDS:
    Kind      <string>
      Kind is a string value representing the REST resource this object
      represents...

    metadata  <Object>
      Standard object's metadata...
    spec      <Object>
          Specification of the desired behavior of the pod...
    status    <Object>
      Most recently observed status of the pod. This data may not be up
      to date...
```

Kubectl выводит объяснение объекта и список атрибутов, которые объект может содержать. Затем можно выполнить детализацию, чтобы узнать о каждом атрибуте больше. Например, можно изучить атрибут spec следующим образом:

```
$ kubectl explain pod.spec
RESOURCE: spec <Object>

DESCRIPTION:
    Specification of the desired behavior of the pod...
    podSpec is a description of a pod.

FIELDS:
  hostPID   <boolean>
    Use the host's pid namespace. Optional: Default to false.
```

```
...

   Volumes    <[]Object>
     List of volumes that can be mounted by containers belonging to the
     pod.

   Containers <[]Object> -required
     List of containers belonging to the pod. Containers cannot currently
     Be added or removed. There must be at least one container in a pod.
     Cannot be updated. More info:
     http://releases.k8s.io/release-1.4/docs/user-guide/containers.md
```

3.2.3 Использование команды kubectl create для создания модуля

Для того чтобы создать модуль из файла YAML, используйте команду `kubectl create`:

```
$ kubectl create -f kubia-manual.yaml
pod "kubia-manual" created
```

Команда `kubectl create -f` применяется для создания любого ресурса (не только модулей) из файла YAML или JSON.

Извлечение полного определения работающего модуля

После создания модуля вы можете запросить у Kubernetes полный YAML модуля. Вы увидите, что он будет похож на YAML, который вы видели ранее. В следующих разделах вы узнаете о дополнительных полях, появляющихся в возвращаемом определении. Для просмотра полного дескриптора модуля выполните следующую ниже команду:

```
$ kubectl get po kubia-manual -o yaml
```

Если вы больше склоняетесь к JSON, то вместо YAML вы также можете поручить `kubectl` вернуть JSON, как тут (это работает, даже если вы использовали YAML для создания модуля):

```
$ kubectl get po kubia-manual -o json
```

Просмотр вновь созданного модуля в списке модулей

Ваш модуль был создан, но как узнать, работает ли он? Давайте выведем список модулей, чтобы увидеть их статусы:

```
$ kubectl get pods

NAME          READY  STATUS    RESTARTS   AGE
```

```
kubia-manual  1/1    Running  0          32s
kubia-zxzij   1/1    Running  0          1d
```

Ваш модуль – kubia-manual. Его статус показывает, что он работает. Если вы похожи на меня, то вы, вероятно, захотите убедиться, что это правда, обменявшись информацией с модулем. Вы сделаете это через минуту. Сначала вы посмотрите на журнал приложения, чтобы свериться с наличием ошибок.

3.2.4 Просмотр журналов приложений

Ваше маленькое приложение Node.js выдает лог в стандартный вывод процесса. Контейнеризированные приложения вместо записи своих логов в файлы обычно выдают лог в стандартный поток вывода и стандартный поток ошибок. Это позволяет пользователям просматривать журналы различных приложений простым и стандартным способом.

Среда выполнения контейнера (Docker в вашем случае) перенаправляет эти потоки в файлы и позволяет получать журнал контейнера, выполнив команду

```
$ docker logs <ид контейнера>
```

Для входа в узел, где работает ваш модуль, и получения его журналов с помощью команды docker logs вы можете использовать ssh, но Kubernetes предоставляет более легкий способ.

Получение журнала модуля с помощью команды kubectl logs

Для того чтобы увидеть журнал модуля (точнее, журнал контейнера), выполните следующую ниже команду на локальной машине (нет необходимости в ssh):

```
$ kubectl logs kubia-manual
Kubia server starting...
```

Вы не отправляли никаких веб-запросов на свое приложение Node.js, поэтому журнал показывает всего одну журнальную инструкцию о запуске сервера. Как видите, если модуль содержит всего один контейнер, получать логи приложения в Kubernetes невероятно просто.

> **ПРИМЕЧАНИЕ.** Журналы контейнера автоматически ротируются ежедневно и всякий раз, когда файл журнала достигает размера 10 Мб. Команда kubectl logs показывает записи журнала только из последней ротации.

Задание имени контейнера при получении логов многоконтейнерного модуля

Если модуль содержит несколько контейнеров, необходимо при выполнении команды kubectl logs явно указать имя контейнера, включив параметр -c <имя контейнера>. В вашем модуле kubia-manual вы устанавливаете имя

контейнера равным kubia, поэтому если в модуле имеются дополнительные контейнеры, то вам нужно будет получить его журналы следующим образом:

```
$ kubectl logs kubia-manual -c kubia
Kubia server starting...
```

Обратите внимание, что можно получать журналы только тех контейнеров модулей, которые все еще существуют. При удалении модуля его журналы также удаляются. Для того чтобы сделать журналы модуля доступными даже после удаления модуля, необходимо настроить централизованное общекластерное ведение журнала, в котором все журналы хранятся в центральном хранилище. Глава 17 объясняет, как работает централизованное ведение журнала.

3.2.5 Отправка запросов в модуль

Модуль сейчас работает – по крайней мере, по сообщениям команды `kubectl get` и журнала вашего приложения. Но как это увидеть в действии? В предыдущей главе команда `kubectl expose` использовалась для создания службы с целью получения доступа к модулю извне. Вы не собираетесь делать это сейчас, потому что целая глава посвящена службам, и у вас есть другие способы подключения к модулю для тестирования и отладки. Один из них – через *переадресацию портов*.

Переадресация локального сетевого порта на порт в модуле

Когда вы хотите обменяться информацией с конкретным модулем без прохождения службы (для отладки или по другим причинам), Kubernetes позволяет настроить переадресацию портов на модуль. Это делается с помощью команды `kubectl port-forward`. Следующая ниже команда перенаправит локальный порт 8888 вашей машины на порт 8080 модуля `kubia-manual`:

```
$ kubectl port-forward kubia-manual 8888:8080
... Forwarding from 127.0.0.1:8888 -> 8080
... Forwarding from [::1]:8888 -> 8080
```

Сервер переадресации портов запущен, и теперь вы можете подключаться к своему модулю через локальный порт.

Подключение к модулю через сервер переадресации портов

В другом терминале теперь можно использовать `curl` для отправки HTTP-запроса в модуль через прокси-сервер переадресации портов `kubectl port-forward`, работающий на `localhost:8888`:

```
$ curl localhost: 8888
You've hit kubia-manual
```

На рис. 3.5 показано слишком упрощенное представление того, что происходит при отправке запроса. На самом деле между процессом `kubectl` и моду-

лем находится несколько дополнительных компонентов, но они не актуальны прямо сейчас.

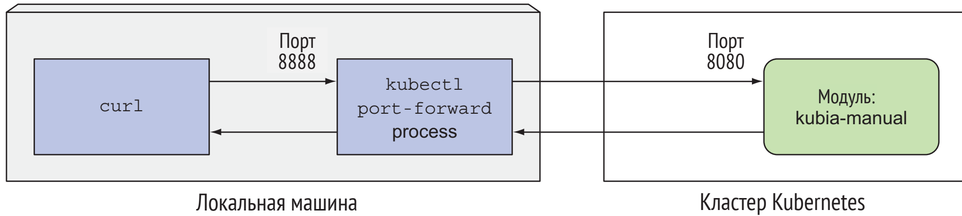


Рис. 3.5. Упрощенное представление того, что происходит при использовании curl с kubectl port-forward

Использование такой переадресации портов является эффективным способом тестирования отдельного модуля. Вы узнаете о других подобных методах далее в книге.

3.3 Организация модулей с помощью меток

На этом этапе в кластере выполняется два модуля. При развертывании реальных приложений большинство пользователей в конечном итоге оказывается в ситуации, когда работает гораздо больше модулей. По мере того как количество модулей увеличивается, потребность в их классификации в подмножества будет становиться все более очевидной.

Например, в случае с архитектурами микросервисов число развернутых микросервисов может легко превысить 20 или больше. Эти компоненты, вероятно, будут реплицированы (будет развернуто несколько копий одного и того же компонента), а несколько версий или выпусков (стабильные, бета, канареечные и т. д.) будут выполняться одновременно. Это может привести к сотням модулей в системе. Без механизма их организации вы получите большой, непонятный беспорядок, подобный показанному на рис. 3.6. На рисунке показаны модули нескольких микросервисов с несколькими работающими репликами, а другие – с разными выпусками одного и того же микросервиса.

Очевидно, что вам нужен способ организации их в небольшие группы на основе произвольных критериев, чтобы каждый разработчик и системный администратор, имеющий дело с вашей системой, мог легко увидеть, какой где модуль. И потребуется обеспечить возможность работы на каждом модуле, принадлежащем определенной группе с помощью одного действия, вместо того чтобы выполнять действие для каждого модуля в отдельности.

Организация модулей и всех других объектов Kubernetes осуществляется с помощью меток.

3.3.1 Знакомство с метками

Метки представляют собой простое, но невероятно мощное функциональное средство Kubernetes для организации не только модулей, но и всех дру-

гих ресурсов Kubernetes. Метка – это произвольная пара «ключ-значение», присоединяемая к ресурсу, которая затем используется при отборе ресурсов с помощью селекторов меток (ресурсы фильтруются на основе того, включают они метку, указанную в селекторе, или нет). Ресурс может иметь несколько меток, если ключи этих меток уникальны в пределах данного ресурса. Метки обычно прикрепляются к ресурсам при их создании, но можно также добавлять дополнительные метки или даже изменять значения существующих меток позже без необходимости повторного создания ресурса.

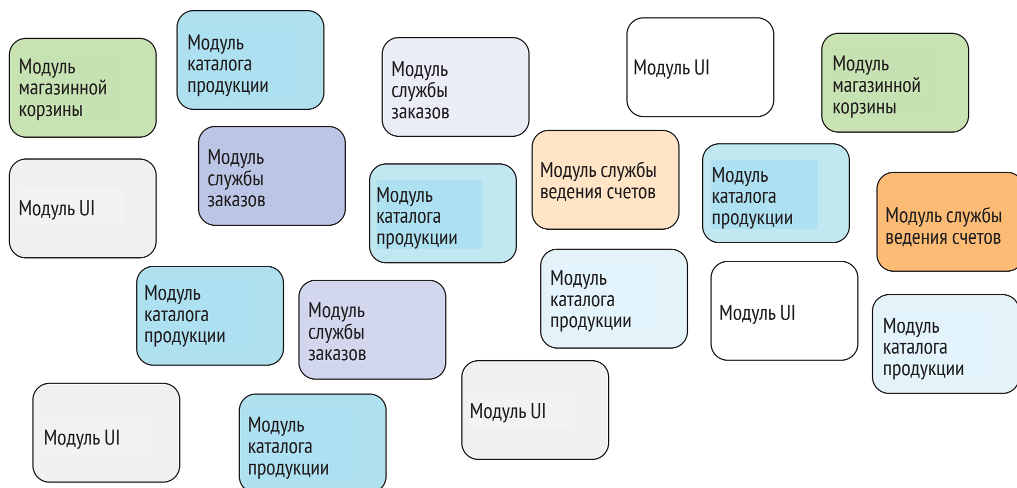


Рис. 3.6. Неклассифицированные модули в микросервисной архитектуре

Вернемся к примеру с микросервисами из рис. 3.6. Добавив ярлыки на эти модули, вы получаете гораздо лучше организованную систему, которую каждый может легко понять. Каждый модуль обозначается с помощью 2 меток:

- `app`, которая указывает, к какому приложению, компоненту или микросервису принадлежит модуль;
- `rel`, которая показывает, является ли приложение, запущенное в модуле, стабильной, бета- или канареечной версией.

ОПРЕДЕЛЕНИЕ. Канареечный релиз – это когда вы развертываете новую версию приложения рядом со стабильной версией, и только небольшая часть пользователей попадает в новую версию, чтобы увидеть, как она себя ведет, прежде чем развернуть ее для всех пользователей. Это препятствует доступу слишком большого числа пользователей к плохим релизам.

Добавив эти две метки, вы, по существу, организовали свои модули в двух измерениях (горизонтально по приложениям и вертикально по релизам), как показано на рис. 3.7.

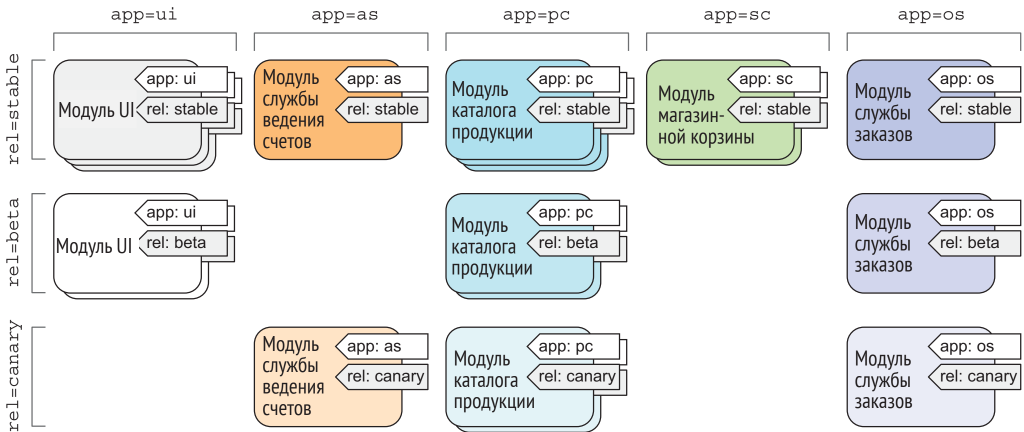


Рис. 3.7. Организация модулей в микросервисной архитектуре с метками модулей

Каждый разработчик или системный администратор с доступом к кластеру теперь, глядя на метки модуля, может легко увидеть структуру системы и то, куда вписывается каждый модуль.

3.3.2 Указание меток при создании модуля

Теперь вы увидите метки в действии, создав новый модуль с двумя метками. Создайте новый файл с именем `kubia-manual-with-labels.yaml` с содержимым следующего ниже листинга.

Листинг 3.3. Модуль с метками: kubia-manual-with-labels.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual
    env: prod
spec:
  containers:
  - image: luksa/kubia
  name: kubia
  ports:
  - containerPort: 8080
    protocol: TCP
```

Две метки прикрепляются к модулю

Вы включили метки `creation_method=manual` и `env= prod`. Теперь вы создадите этот модуль:

```
$ kubectl create -f kubia-manual-with-labels.yaml
pod "kubia-manual-v2" created
```

Команда `kubectl get pods` по умолчанию не выводит никаких меток, но их можно увидеть с помощью параметра `--show-labels`:

```
$ kubectl get po --show-labels
NAME            READY  STATUS   RESTARTS  AGE  LABELS
kubia-manual    1/1    Running  0         16m  <none>
kubia-manual-v2 1/1    Running  0         2m   creat_method=manual,env=prod
kubia-zxzij     1/1    Running  0         1d   run=kubia
```

Если вас интересуют только определенные метки, то вместо вывода всех меток вы можете указать их с помощью переключателя `-L` и отобразить каждую из них в своем собственном столбце. Снова выведем список модулей и покажем столбцы для двух меток, которые вы прикрепили к своему модулю `kubia-manual-v2`:

```
$ kubectl get po -L creation_method,env
NAME            READY  STATUS   RESTARTS  AGE  CREATION_METHOD  ENV
kubia-manual    1/1    Running  0         16m  <none>           <none>
kubia-manual-v2 1/1    Running  0         2m   manual           prod
kubia-zxzij     1/1    Running  0         1d   <none>           <none>
```

3.3.3 Изменение меток существующих модулей

Метки можно также добавлять и модифицировать на существующих модулях. Поскольку модуль `kubia-manual` также был создан вручную, давайте добавим к нему метку `creation_method=manual`:

```
$ kubectl label po kubia-manual creation_method=manual
pod "kubia-manual" labeled
```

Теперь давайте поменяем метку `env=prod` на `env=debug` на модуле `kubia-manual-v2`, чтобы увидеть, как можно изменять существующие метки.

> **ПРИМЕЧАНИЕ.** При изменении существующих меток нужно использовать параметр `--overwrite`.
>
> ```
> $ kubectl label po kubia-manual-v2 env=debug --overwrite
> pod "kubia-manual-v2" labeled
> ```

Выведем список модулей еще раз, для того чтобы увидеть обновленные метки:

```
$ kubectl get po -L creation_method,env
NAME             READY  STATUS   RESTARTS  AGE  CREATION_METHOD  ENV
kubia-manual     1/1    Running  0         16m  manual           <none>
kubia-manual-v2  1/1    Running  0         2m   manual           debug
kubia-zxzij      1/1    Running  0         1d   <none>           <none>
```

Как вы можете видеть, прикрепление меток к ресурсам не составляет труда, и таким же легким является их изменение на существующих ресурсах. Прямо сейчас это может быть не очевидно, но это невероятно мощное функциональное средство, и вы в этом убедитесь в следующей главе. Но сначала давайте посмотрим, что вы можете сделать с этими метками в дополнение к их отображению на экране при выводе списка модулей.

3.4 Перечисление подмножеств модулей посредством селекторов меток

Само прикрепление меток к ресурсам, чтобы вы могли видеть метки рядом с каждым ресурсом при выводе их списка, не так интересно. Но метки идут рука об руку с *селекторами меток*. Селекторы меток позволяют выбрать подмножество модулей, помеченных определенными метками, и выполнять операцию на этих модулях. Селектор меток – это критерий, который фильтрует ресурсы на основе того, содержат они определенную метку с определенным значением или нет.

Селектор меток может выбирать ресурсы в зависимости от того:

- содержит ли (или не содержит) ресурс метку с определенным ключом;
- содержит ли ресурс метку с определенным ключом и значением;
- содержит ли ресурс метку с определенным ключом, но со значением, не равным указанному вами.

3.4.1 Вывод списка модулей с помощью селектора меток

Давайте применим селекторы меток на модулях, которые вы создали к этому моменту. Для того чтобы просмотреть все модули, созданные вручную (вы пометили их `creation_method=manual`), выполните следующие ниже действия:

```
$ kubectl get po -l creation_method=manual
NAME              READY   STATUS    RESTARTS   AGE
kubia-manual      1/1     Running   0          51m
kubia-manual-v2   1/1     Running   0          37m
```

Для того чтобы вывести список всех модулей, которые включают метку `env`, каким бы ни было его значение:

```
$ kubectl get po -l env
NAME              READY   STATUS    RESTARTS   AGE
kubia-manual-v2   1/1     Running   0          37m
```

И те, которые не имеют метки `env`:

```
$ kubectl get po -l '!env'
NAME              READY   STATUS    RESTARTS   AGE
```

```
kubia-manual     1/1    Running  0         51m
kubia-zxzij      1/1    Running  0         10d
```

> **ПРИМЕЧАНИЕ.** Не забудьте применить одинарные кавычки вокруг `!env`, чтобы оболочка bash не вычислила восклицательный знак.

Аналогичным образом можно также сопоставить модули со следующими ниже селекторами меток:

- `creation_method!=manual`, чтобы выбрать модули с меткой `creation_method` с любым значением, кроме `manual`;
- `env in (prod,devel)`, чтобы выбрать модули с меткой `env`, установленной в `prod` или `development`;
- `env notin (prod,devel)`, чтобы выбрать модули с меткой `env`, установленной в любое значение, кроме `prod` или `devel`.

Возвращаясь к модулям в примере микросервисно-ориентированной архитектуры, все модули, входящие в состав микросервиса каталога продукции, можно выбрать с помощью селектора меток `app=pc` (показано на следующем ниже рисунке).

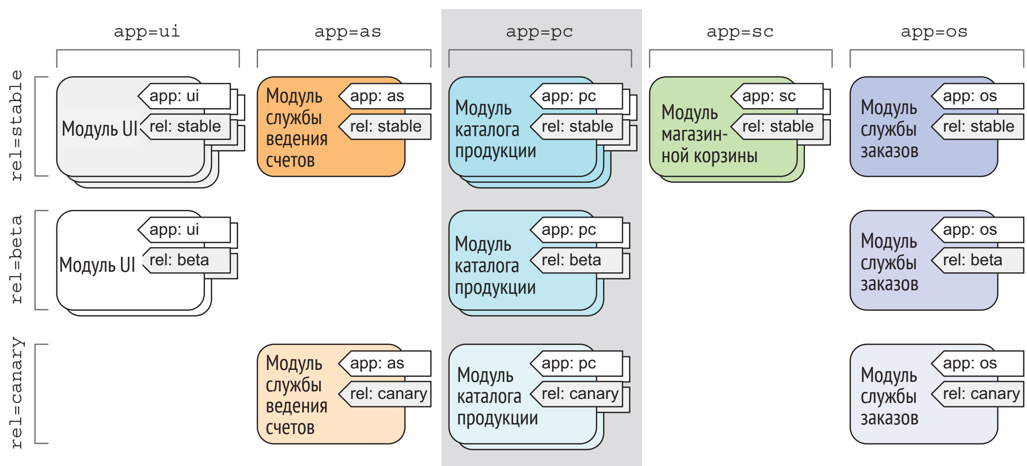


Рис. 3.8. Выбор модулей микросервисов с каталогами продукции с помощью селектора меток "app=pc"

3.4.2 Использование нескольких условий в селекторе меток

Селектор может также включать несколько критериев, разделенных запятыми. Для того чтобы соответствовать селектору, ресурсы должны соответствовать всем меткам. Если, например, вы хотите выбрать только pc, на которых выполняется бета-версия микросервиса каталога продукции, примените следующий селектор: `app=pc, rel=beta` (визуализировано на рис. 3.9).

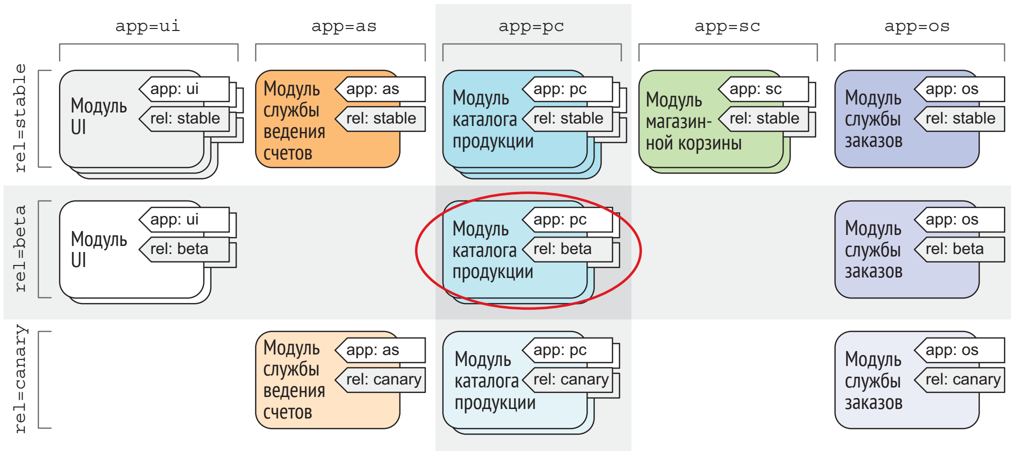


Рис. 3.9. Выбор модулей с несколькими селекторами меток

Селекторы меток полезны не только для выведения списка модулей, но и для выполнения действий на подмножестве всех модулей. Например, далее в этой главе вы увидите, как использовать селекторы меток для удаления нескольких модулей одновременно. Вместе с тем селекторы меток используются не только инструментом командной строки `kubectl`. Они также используются внутри, как вы увидите далее.

3.5 Использование меток и селекторов для ограничения планирования модулей

Все модули, которые вы создали до сих пор, были запланированы на ваших рабочих узлах в значительной степени случайным образом. Как я уже упоминал в предыдущей главе, это правильный способ работы в кластере Kubernetes. Поскольку Kubernetes обеспечивает доступ ко всем узлам в кластере как к одной большой платформе развертывания, для вас не имеет значения, на какой узел запланирован модуль. Так как каждый модуль получает точное количество вычислительных ресурсов, которые он запрашивает (процессор, память и т. д.), и его доступность из других модулей вообще не зависит от узла, на который запланирован модуль, обычно вам не нужно указывать Kubernetes, где именно планировать свои модули.

Однако существуют определенные случаи, когда вы захотите иметь хотя бы небольшое право голоса в том, где должен быть запланирован модуль. Хороший пример – когда ваша аппаратная инфраструктура неоднородна. Если часть рабочих узлов имеет вращающиеся (магнитные) жесткие диски, в то время как другие имеют твердотельные накопители (SSD), можно запланировать определенные модули для одной группы узлов, а остальные – для другой. Еще один пример – когда необходимо приписать модули, выполняющие интенсивные вычисления на основе графического процессора (GPU), только к узлам, которые предоставляют необходимое ускорение GPU.

Вам никогда не следует заявлять конкретно, на какой узел должен быть запланирован модуль, потому что это привяжет приложение к инфраструктуре, в то время как вся идея Kubernetes состоит в скрытии фактической инфраструктуры от приложений, которые работают на ней. Но если вы действительно хотите сказать, где должен быть запланирован модуль, то, вместо того чтобы указывать точный узел, вы должны описать требования к узлу, а затем позволить Kubernetes выбрать узел, который соответствует этим требованиям. Это можно сделать с помощью меток узлов и селекторов меток узлов.

3.5.1 Использование меток для классификации рабочих узлов

Как вы узнали ранее, модули не являются единственным типом ресурсов Kubernetes, к которому можно прикреплять метку. Метки могут быть присоединены к любому объекту Kubernetes, включая узлы. Обычно, когда системные администраторы добавляют новый узел в кластер, они классифицируют узел, прикрепляя метки, определяющие тип оборудования, которое предоставляет узел, или что-то еще, что может пригодиться при планировании модулей.

Предположим, что один из узлов кластера содержит GPU, предназначенный для использования в вычислениях общего назначения. Вы хотите добавить метку в узел, показав эту его особенность. Вы добавляете метку `gpu=true` в один из ваших узлов (выберите один из списка, возвращенного `kubectl get nodes`):

```
$ kubectl label node gke-kubia-85f6-node-0rrx gpu=true
node "gke-kubia-85f6-node-0rrx" labeled
```

Теперь вы можете использовать селектор меток при выведении списка узлов, как это было раньше с модулями. Выведем список только тех узлов, которые содержат метку `gpu=true`:

```
$ kubectl get nodes -l gpu=true
NAME                      STATUS  AGE
gke-kubia-85f6-node-0rrx  Ready   1d
```

Как и ожидалось, эту метку имеет всего один узел. Можно также попробовать вывести список всех узлов и указать `kubectl` отобразить дополнительный столбец со значениями метки `gpu` каждого узла (`kubectl get nodes -L gpu`).

3.5.2 Приписывание модулей к определенным узлам

Теперь представьте, что вы хотите развернуть новый модуль, который для выполнения своей работы нуждается в GPU. Для того чтобы попросить планировщик выбирать только те узлы, которые предоставляют GPU, добавьте селектор узлов в YAML модуля. Создайте файл с именем `kubia-gpu.yaml` со следующим содержимым листинга и затем примените `kubectl create-f kubia-gpu.yaml` для создания модуля.

Листинг 3.4. Использование селектора меток для планирования модуля на определенный узел: kubia-gpu.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
  nodeSelector:
    gpu: "true"
  containers:
  - image: luksa/kubia
    name: kubia
```

nodeSelector сообщает Kubernetes разворачивать этот модуль только на узлы, содержащие метку gpu=true

Вы добавили поле `nodeSelector` в секцию спецификации `spec`. При создании модуля планировщик будет выбирать только те узлы, которые содержат метку `gpu=true` (в вашем случае это всего один узел).

3.5.3 Планирование размещения на один конкретный узел

Точно так же можно приписать модуль конкретному узелу, потому что каждый узел тоже имеет уникальную метку с ключом `kubernetes.io/hostname` и значение, установленное в фактическое хост-имя узла. Вместе с тем, в случае если узел отключен, установка поля `nodeSelector` в конкретный узел по метке хост-имени может привести к тому, что модуль будет непригодным для приписывания (планирования). Вы не должны думать с точки зрения отдельных узлов. Всегда думайте о логических группах узлов, удовлетворяющих определенным критериям, заданным с помощью селекторов меток.

Это была быстрая демонстрация того, как работают метки и селекторы меток и как их можно использовать для влияния на работу системы Kubernetes. Важность и полезность селекторов меток станет еще более очевидной, когда мы поговорим о контроллерах репликации и службах в следующих двух главах.

ПРИМЕЧАНИЕ. Дополнительные способы воздействия на то, к какому узлу запланировано использование модуля, рассматриваются в главе 16.

3.6 Аннотирование модулей

Помимо меток, модули и другие объекты также могут содержать *аннотации*. Аннотации тоже являются парами ключ-значение, поэтому, по сути, они похожи на метки, но не предназначены для хранения идентифицирующей информации. Они не могут использоваться для группирования объектов, так же как метки. В то время как объекты можно выбирать с помощью селекторов меток, селектора аннотаций не существует.

С другой стороны, аннотации могут содержать гораздо большие фрагменты информации и в первую очередь предназначены для использования утилитами. Некоторые аннотации автоматически добавляются в объекты с помощью Kubernetes, а другие добавляются пользователями вручную.

Аннотации используются при внесении новых функциональных возможностей в Kubernetes. Как правило, альфа- и бета-версии новых функциональных возможностей не вводят никаких новых полей в объекты API. Вместо полей используются аннотации, а затем, как только необходимые изменения API становятся ясными и согласоваными среди разработчиков Kubernetes, вводятся новые поля, и соответствующие аннотации устаревают.

Замечательное применение аннотаций – это добавление описаний для каждого модуля или другого объекта API, чтобы каждый, кто использует кластер, мог быстро найти информацию о каждом отдельном объекте. Например, аннотация, используемая для указания имени пользователя, создавшего объект, может значительно упростить взаимодействие между всеми, кто работает в кластере.

3.6.1 Поиск аннотаций объекта

Рассмотрим пример аннотации, которую система Kubernetes автоматически добавила в модуль, созданный в предыдущей главе. Для просмотра аннотаций необходимо запросить полный YAML модуля или использовать команду `kubectl describe`. В следующем ниже листинге вы примените первый вариант.

Листинг 3.5. Аннотации модуля

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind":"SerializedReference", "apiVersion":"v1",
      "reference":{"kind":"ReplicationController", "namespace":"default", ...
```

Не вдаваясь в слишком много деталей, как вы можете видеть, аннотация `kubernetes.io/created-by` содержит данные JSON об объекте, который создал модуль. Это не то, что вы хотели бы поместить в метку. Метки должны быть короткими, в то время как аннотации могут содержать относительно большие двоичные блобы данных (в общей сложности до 256 Кб).

> **ПРИМЕЧАНИЕ.** Аннотации `kubernetes.io/created-by` устарели в версии 1.8 и будут удалены в 1.9, так что вы больше не увидите их в YAML.

3.6.2 Добавление и изменение аннотаций

Аннотации, очевидно, могут добавлятся в модули во время создания, так же как и метки. Их можно также добавлять или модифицировать на существую-

щих модулях позднее. Самый простой способ добавить аннотацию в существующий объект – применить команду `kubectl annotate`. Сейчас вы попробуете добавить аннотацию в ваш модуль `kubia-manual`:

```
$ kubectl annotate pod kubia-manual mycompany.com/someannotation="foo bar"
pod "kubia-manual" annotated
```

Вы добавили аннотацию `mycompany.com/someannotation` со значением `foo bar`. Для ключей аннотаций рекомендуется использовать этот формат, чтобы предотвратить конфликты ключей. Когда различные инструменты или библиотеки добавляют аннотации в объекты, они могут случайно переопределить аннотации друг друга, если в них не используются уникальные префиксы, как вы сделали здесь.

Для просмотра добавленной аннотации можно применить команду `kubectl describe`:

```
$ kubectl describe pod kubia-manual
...
Annotations: mycompany.com/someannotation=foo bar
...
```

3.7 Использование пространств имен для группирования ресурсов

Давайте на минутку вернемся к меткам. Мы видели, как они организуют в группы модули и другие объекты. Поскольку каждый объект может иметь несколько меток, эти группы объектов могут перекрываться. Кроме того, если во время работы с кластером (например, посредством `kubectl`) явно не указать селектор меток, то всегда будут отображаться все объекты.

А как насчет случаев, когда вы хотите разбить объекты на отдельные, неперекрывающиеся группы? Вы можете захотеть работать только внутри одной из групп в данный момент времени. По этой и другим причинам Kubernetes также группирует объекты в пространства имен. Это не пространства имен Linux, о которых мы говорили в главе 2, которые используются для изоляции процессов друг от друга. Пространства имен Kubernetes предоставляют область видимости для имен объектов. Вместо того чтобы иметь все ресурсы в одном пространстве имен, вы можете разбить их на несколько пространств имен, что также позволяет использовать одни и те же имена ресурсов несколько раз (в разных пространствах имен).

3.7.1 Необходимость пространств имен

Использование нескольких пространств имен позволяет разбить сложные системы со множеством компонентов на более мелкие отдельные группы. Они также могут использоваться для разделения ресурсов в мультитенантной среде, разбивая ресурсы на среды рабочего окружения, среды разработ-

ки и среды контроля качества либо любым другим способом. Имена ресурсов должны быть уникальными только в пределах пространства имен. Два разных пространства имен могут содержать ресурсы с одинаковыми именами. Но хотя большинство типов ресурсов организовано в пространства имен, некоторые из них – нет. Одним из них является ресурс узла (Node), который является глобальным и не привязан к одному пространству имен. О других ресурсах кластерного уровня вы узнаете в последующих главах.

Давайте посмотрим, как использовать пространства имен.

3.7.2 Обнаружение других пространств имен и их модулей

Прежде всего давайте выведем список всех пространств имен в кластере:

```
$ kubectl get ns
NAME         LABELS  STATUS  AGE
default      <none>  Active  1h
kube-public  <none>  Active  1h
kube-system  <none>  Active  1h
```

До этого момента вы работали только в пространстве имен `default`. При выведении списка ресурсов с помощью команды `kubectl get` вы никогда не указывали пространство имен явным образом, поэтому `kubectl` всегда устанавливал пространство имен `default`, показывая вам объекты только в этом пространстве. Но, как видно из списка, существуют также пространства имен `kube-public` и `kube-system`. Давайте посмотрим на модули, которые принадлежат пространству имен `kube-system`, поручив `kubectl` вывести список модулей только в этом пространстве имен:

```
$ kubectl get po --namespace kube-system
NAME                                READY  STATUS   RESTARTS  AGE
fluentd-cloud-kubia-e8fe-node-txje  1/1    Running  0         1h
heapster-v11-fz1ge                  1/1    Running  0         1h
kube-dns-v9-p8a4t                   0/4    Pending  0         1h
kube-ui-v4-kdlai                    1/1    Running  0         1h
l7-lb-controller-v0.5.2-bue96       2/2    Running  92        1h
```

СОВЕТ. Вместо параметра `--namespace` можно также использовать параметр `-n`.

Вы узнаете об этих модулях позже из книги (не переживайте, если модули, показанные здесь, не совпадают в точности с теми, которые имеются в вашей системе). Из имени пространства имен ясно, что эти ресурсы связаны с самой системой Kubernetes. Размещение их в этом отдельном пространстве имен позволяет ей содержать все в организованном порядке. Если бы все они были в пространстве имен по умолчанию (default), они были бы перемешаны с ресурсами, которые вы создаете сами, и тогда вам было бы трудно увидеть, что принадлежит чему, и вы могли бы случайно удалить системные ресурсы.

Пространства имен позволяют разделить ресурсы, которые не принадлежат друг другу, на неперекрывающиеся группы. Если несколько пользователей или групп пользователей используют один кластер Kubernetes и каждый из них управляет своим собственным набором ресурсов, каждый из них должен использовать собственное пространство имен. Благодаря этому им не нужно проявлять особую осторожность, чтобы случайно не изменить или не удалить ресурсы других пользователей, и не нужно беспокоиться о конфликтах имен, потому что, как уже отмечалось, пространства имен предоставляют область действия для имен ресурсов.

Помимо изоляции ресурсов, пространства имен также используются для предоставления доступа к определенным ресурсам только определенным пользователям и даже для ограничения количества вычислительных ресурсов, доступных отдельным пользователям. Вы узнаете об этом в главах с 12 по 14.

3.7.3 Создание пространства имен

Пространство имен – это ресурс Kubernetes, как и любой другой, поэтому его можно создать, разместив файл YAML на сервере API Kubernetes. Давайте сейчас посмотрим, как это сделать.

Создание пространства имен из файла yaml

Сначала создайте файл для своего собственного пространства имен `custom-namespace.yaml` с листингом со следующим ниже содержимым (вы найдете файл в архиве кода, прилагаемого к этой книге).

Листинг 3.6. Определение YAML пространства имен: custom-namespace.yaml

```
apiVersion: v1
kind: Namespace             <-- Это сообщает, что вы задаете пространство имен
metadata:
  name: custom-namespace    <-- Это имя пространства имен
```

Теперь примените `kubectl`, чтобы отправить файл на сервер API Kubernetes:

```
$ kubectl create -f custom-namespace.yaml
namespace "custom-namespace" created
```

Создание пространства имен с помощью команды kubectl create namespace

Хотя написание такого файла, как приведенный выше, – это не такое уж и большое дело, все же оно хлопотное. К счастью, вы также можете создавать пространства имен с помощью специальной команды `kubectl create namespace`, которая быстрее, чем написание файла YAML. Заставив вас создать манифест YAML для пространства имен, я хотел укрепить идею о том, что все в Kubernetes имеет соответствующий объект API, который можно создавать, читать, обновлять и удалять, отправляя манифест YAML на сервер API.

Вы могли бы создать пространство имен следующим образом:

```
$ kubectl create namespace custom-namespace
namespace "custom-namespace" created
```

> **ПРИМЕЧАНИЕ.** Хотя большинство имен объектов должно соответствовать соглашениям об именах, указанных в RFC 1035 (доменные имена), что означает, что они могут содержать только буквы, цифры, тире и точки, в пространствах имен (и некоторых других) не допускается наличие точек.

3.7.4 Управление объектами в других пространствах имен

Для того чтобы создать ресурсы в созданном вами пространстве имен, добавьте запись `namespace: custom-namespace` в секцию `metadata` или укажите пространство имен при создании ресурса с помощью команды `kubectl create`:

```
$ kubectl create -f kubia-manual.yaml -n custom-namespace
pod "kubia-manual" created
```

Теперь у вас есть два модуля с одинаковым именем (`kubia-manual`). Один находится в пространстве имен `default`, а другой – в вашем `custom-namespace`.

При выводе списка, описании, изменении или удалении объектов в других пространствах имен необходимо передать в `kubectl` флаг `--namespace` (или `-n`). Если пространство имен не указано, то `kubectl` выполняет действие в пространстве имен по умолчанию (default), настроенном в текущем контексте `kubectl`. Пространство имен текущего контекста и сам текущий контекст можно изменить с помощью команд `kubectl config`. Для того чтобы узнать больше об управлении контекстами `kubectl`, обратитесь к приложению A.

> **СОВЕТ.** Для быстрого переключения на другое пространство имен можно настроить следующий псевдоним: `alias kcd='kubectl config set-context $(kubectl config currentcontext) --namespace '`. Затем можно переключаться между пространствами имен с помощью `kcd some-namespace`.

3.7.5 Изоляция, обеспечиваемая пространствами имен

Для того чтобы завершить этот раздел о пространствах имен, позвольте объяснить, что пространства имен не обеспечивают – по крайней мере, не из коробки. Хотя пространства имен позволяют изолировать объекты в различные группы, что позволяет работать только с теми из них, которые принадлежат указанному пространству имен, они не обеспечивают какой-либо изоляции запущенных объектов.

Например, можно подумать, что когда разные пользователи развертывают модули в разных пространствах имен, эти модули изолированы друг от друга и не могут взаимодействовать. Однако это не обязательно так. Наличие в про-

странствах имен сетевой изоляции зависит от того, какое решение для работы с сетью внутри Kubernetes. Если в ситуации, когда решение не обеспечивает сетевую изоляцию между пространствами имен, модуль в пространстве имен `foo` знает IP-адрес модуля в пространства имен `bar`, ничто не мешает ему отправлять трафик, например HTTP-запросы, в другой модуль.

3.8 Остановка и удаление модулей

Вы создали несколько модулей, которые все до сих пор работают. У вас работает четыре модуля в пространстве имен `default` и один модуль в пространстве имен `custom-namespace`. Теперь вы остановите их все, потому что они вам больше не нужны.

3.8.1 Удаление модуля по имени

Сначала удалите модуль `kubia-gpu` по имени:

```
$ kubectl delete po kubia-gpu
pod "kubia-gpu" deleted
```

Удаляя модуль, вы поручаете Kubernetes завершить работу всех контейнеров, входящих в этот модуль. Kubernetes посылает сигнал `SIGTERM` процессу и ожидает определенное количество секунд (по умолчанию 30) для корректного завершения работы. Если он не выключается вовремя, процесс затем убивается через `SIGKILL`. Для того чтобы убедиться, что ваши процессы всегда корректно завершают работу, они должны правильно обрабатывать сигнал `SIGTERM`.

> **СОВЕТ.** Можно также удалить несколько модулей, указав несколько имен, разделенных пробелами (например, `kubectl delete po pod1 pod2`).

3.8.2 Удаление модулей с помощью селекторов меток

Вместо указания каждого удаляемого модуля по имени, для того чтобы остановить оба модуля: и `kubia-manual`, и `kubia-manual-v2`, – вы теперь будете использовать то, что вы узнали о селекторах меток. Оба модуля содержат метку `creation_method=manual`, поэтому их можно удалить с помощью селектора меток:

```
$ kubectl delete po -l creation_method=manual
pod "kubia-manual" deleted
pod "kubia-manual-v2" deleted
```

В более раннем примере с микросервисами, где у вас были десятки (или, возможно, сотни) модулей, вы могли бы, например, удалить все канареечные модули за один раз, указав селектор меток `rel=canary` (визуализировано на рис. 3.10):

```
$ kubectl delete po -l rel=canary
```

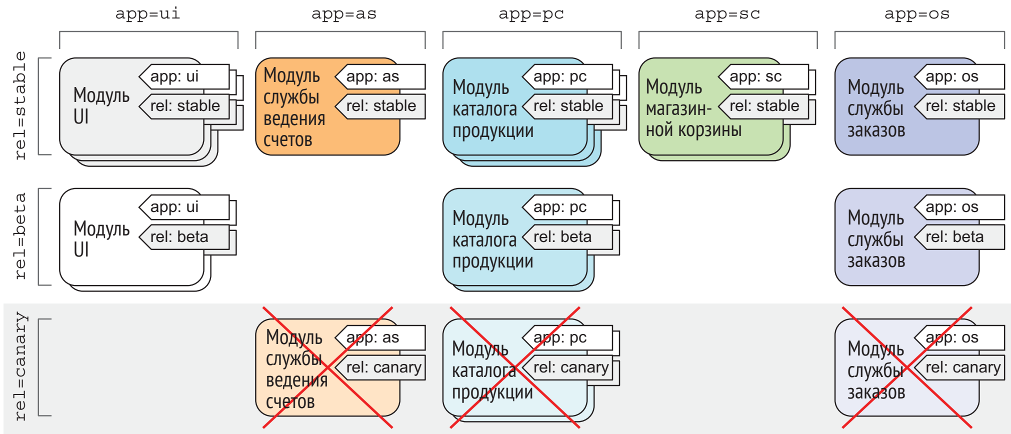


Рис. 3.10. Выбор и удаление всех канареечных модулей посредством метки rel=canary

3.8.3 Удаление модулей путем удаления всего пространства имен

Хорошо, но возвращаемся к вашим настоящим модулям. Как насчет модуля в пространстве имен `custom-namespace`? Вам больше не нужны ни модули в этом пространстве имен, ни само пространство имен. Вы можете удалить все пространство имен (модули будут удалены автоматически вместе с пространством имен), используя следующую команду:

```
$ kubectl delete ns custom-namespace
namespace "custom-namespace" deleted
```

3.8.4 Удаление всех модулей в пространстве имен при сохранении пространства имен

Теперь вы очистили почти все. Но как насчет модуля, созданного с помощью команды `kubectl run` в главе 2? Он по-прежнему работает:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-zxzij   1/1     Running   0          1d
```

На этот раз вместо удаления определенного модуля сообщите Kubernetes удалить все модули в текущем пространстве имен с помощью параметра `--all`:

```
$ kubectl delete po --all
pod "kubia-zxzij" deleted
```

Теперь перепроверьте, чтобы ни один модуль не был оставлен в рабочем состоянии:

```
$ kubectl get pods
NAME         READY  STATUS       RESTARTS  AGE
kubia-09as0  1/1    Running      0         1d
kubia-zxzij  1/1    Terminating  0         1d
```

Постойте, что?! Модуль `kubia-zxzij` завершается, но появился новый модуль под названием `kubia-09as0`, которого раньше не было. Не важно, сколько раз вы удалите все модули, все равно будет появляться новый под названием *kubia-что-то*.

Вы, возможно, помните, что вы создали свой первый модуль командой `kubectl run`. В главе 2 я отметил, что она не создает модуль непосредственно, а вместо этого создает контроллер репликации, который затем создает модуль. Как только вы удаляете модуль, созданный контроллером репликации, он сразу же создает новый. Для того чтобы удалить модуль, необходимо также удалить контроллер репликации.

3.8.5 Удаление (почти) всех ресурсов в пространстве имен

Вы можете удалить контроллер репликации и модули, а также все созданные службы, удалив все ресурсы в текущем пространстве имен с помощью одной команды:

```
$ kubectl delete all --all
pod "kubia-09as0" deleted
replicationcontroller "kubia" deleted
service "kubernetes" deleted
service "kubia-http" deleted
```

Первый параметр `all` в данной команде указывает, что вы удаляете ресурсы всех типов, а параметр `--all` указывает, что вы удаляете все экземпляры ресурсов, вместо того чтобы указывать их по имени (этот параметр уже использовался при выполнении предыдущей команды удаления).

> **ПРИМЕЧАНИЕ.** Удаление с использованием ключевого слова `all` не удаляет абсолютно все. Некоторые ресурсы (например, секреты, которые мы введем в главе 7) сохраняются и должны быть удалены явным образом.

При удалении ресурсов `kubectl` будет печатать имена всех удаляемых ресурсов. В списке вы должны увидеть контроллер репликций `kubia` и службу `kubia-http`, созданные в главе 2.

> **ПРИМЕЧАНИЕ.** Команда `kubectl delete all --all` также удаляет службу `kubernetes`, но она должна быть автоматически воссоздана через несколько минут.

3.9 Резюме

После прочтения этой главы вы должны теперь иметь приличное понимание центрального строительного блока в Kubernetes. Каждая другая концепция, о которой вы узнаете в последующих нескольких главах, напрямую связана с модулями.

В этой главе вы узнали:

- как решить, должны ли определенные контейнеры быть сгруппированы вместе в модуле или нет;
- модули могут выполнять несколько процессов и аналогичны физическим хостам в неконтейнерном мире;
- дескрипторы YAML или JSON могут быть написаны и использованы для создания модулей, а затем исследованы, чтобы увидеть спецификацию модуля и его текущее состояние;
- метки и селекторы меток следует использовать для организации модулей и упрощения выполнения операций с несколькими модулями одновременно;
- метки и селекторы узлов можно использовать, чтобы назначать модули только тем узлам, которые имеют определенные функциональные особенности;
- аннотации позволяют прикреплять более крупные блобы данных к модулям, и это делается либо пользователями, либо инструментами и библиотеками;
- пространства имен можно использовать для того, чтобы разные группы специалистов могли использовать тот же кластер, как если бы они использовали разные кластеры Kubernetes;
- как использовать команду `kubectl explain` для быстрого поиска информации на любом ресурсе Kubernetes.

В следующей главе вы узнаете о контроллерах репликации и других ресурсах, которые управляют модулями.

Глава 4

Контроллер репликации и другие контроллеры: развертывание управляемых модулей

Эта глава посвящена:

- поддержанию модулей в здоровом состоянии;
- запуску нескольких экземпляров одного модуля;
- автоматическому перепланированию модулей после аварийного завершения работы узла;
- горизонтальному масштабированию модулей;
- запуску модулей системного уровня на каждом узле кластера;
- выполнению пакетных заданий;
- планированию заданий для периодического или однократного выполнения в будущем.

Как вы узнали к этому моменту, модули представляют собой основную развертываемую единицу в Kubernetes. Вы знаете, как создавать, контролировать и управлять ими вручную. Но в случаях реального использования требуется, чтобы ваши развертывания автоматически поддерживались в рабочем состоянии и оставались здоровыми без какого-либо ручного вмешательства. Для этого модули почти никогда не создаются напрямую. Вместо этого создаются другие типы ресурсов, такие как контроллеры репликации (ReplicationController) или развертывания (Deployment), которые затем создают фактические модули и управляют ими.

При создании неуправляемых модулей (например, созданных в предыдущей главе) для запуска модуля выбирается узел кластера, а затем его контейнеры запускаются на этом узле. В этой главе вы узнаете, что Kubernetes затем отслеживает эти контейнеры и автоматически перезапускает их, если они аварийно завершают работу. Вместе с тем, если весь узел завершает работу аварийно, то при условии что этими модулями не управляют ранее упомянутые контроллеры репликации или подобные им, модули на узле будут потеряны и не будут заменены новыми. В этой главе вы узнаете, как Kubernetes проверяет, жив ли контейнер, и перезапускает его, если это не так. Вы также узнаете, как запускать управляемые модули – как те, которые работают бесконечно, так и те, которые выполняют одну задачу, а затем останавливаются.

4.1 Поддержание модулей в здоровом состоянии

Одним из основных преимуществ использования Kubernetes является его способность поддерживать работу списка контейнеров, расположенных где-то в кластере. Это можно сделать, создав ресурс модуля (Pod) и позволив Kubernetes выбрать для него рабочий узел и запускать контейнеры модуля на этом узле. Но что, если один из этих контейнеров умрет? Что делать, если все контейнеры модуля умрут?

Как только модуль назначен узлу, агент Kubelet на том самом узле запустит его контейнеры и с того момента будет поддерживать их в рабочем состоянии, пока модуль существует. Если происходит аварийный сбой главного процесса контейнера, Kubelet перезапустит контейнер. Если ваше приложение имеет ошибку, которая время от времени приводит к его падению, Kubernetes автоматически его перезапустит, поэтому, даже не делая ничего особенного в самом приложении, запуск приложения в Kubernetes автоматически придает ему способность самоисцелиться.

Но иногда приложения перестают работать без падения их процесса. Например, приложение Java с утечкой памяти начнет выдавать ошибки из-за нехватки оперативной памяти, но процесс JVM будет продолжать работать. Было бы здорово иметь способ, которым приложение подавало бы сигналы системе Kubernetes, что оно больше не функционирует как надо, и тем самым вынуждало бы Kubernetes его перезапускать.

Мы сказали, что сбойный контейнер перезапускается автоматически, поэтому, возможно, вы думаете, что вы могли бы отловить эти типы ошибок в приложении и выйти из процесса, когда они происходят. Вы, конечно, можете это сделать, но это все равно не решает всех ваших проблем.

Например, как быть в ситуациях, когда приложение перестает отвечать на запросы из-за бесконечного цикла или взаимоблокировки? Для того чтобы убедиться, что приложения перезапущены, в таких случаях необходимо проверить работоспособность приложения извне и не зависеть от приложения, выполняющего это внутренне.

4.1.1 Знакомство с проверками живучести

Kubernetes может проверить текущую работоспособность контейнера посредством *проверок живучести* (livenessProbe). Для каждого контейнера в спецификации модуля можно указать проверку живучести. Kubernetes будет периодически выполнять проверку и перезапускать контейнер в случае несработки проверки.

> **ПРИМЕЧАНИЕ.** Kubernetes также поддерживает *проверки готовности* (readinessProbe), о которых мы узнаем в следующей главе. Убедитесь, что не путаете одну с другой. Они используются для двух разных целей.

Kubernetes может исследовать контейнер с помощью одного из трех механизмов:

- проверка *HTTP GET* выполняет запрос HTTP GET на IP-адрес, порт и путь контейнера, которые вы укажете. Если проверка получает отклик и код ответа не представляет ошибку (другими словами, если код отклика HTTP будет 2xx или 3xx), проверка считается сработавшей. Если сервер возвращает отклик с кодом ошибки или вообще не отвечает, то проверка считается несработавшей, и в результате контейнер будет перезапущен;
- проверка *сокета TCP* пытается открыть TCP-подключение к указанному порту контейнера. Если подключение установлено успешно, то проверка сработала. В противном случае контейнер перезапускается;
- проверка *Exec* выполняет произвольную команду внутри контейнера и проверяет код состояния на выходе из команды. Если код состояния равен 0, то проверка выполнена успешно. Все остальные коды считаются несработавшими.

4.1.2 Создание проверки живучести на основе HTTP

Давайте посмотрим, как добавить проверку живучести в ваше приложение Node.js. Поскольку мы имеем дело с веб-приложением, имеет смысл добавить проверку живучести, которая будет проверять, обслуживает ли запросы веб-сервер приложения. Но поскольку это конкретное приложение Node.js слишком простое, чтобы когда-либо потерпеть аварийный сбой, вам нужно заставить это приложение аварийно завершиться искусственным образом.

Для того чтобы правильно продемонстрировать проверку живучести, вы немного видоизмените приложение и заставите его возвращать код состояния HTTP 500 с внутренней ошибкой сервера для каждого запроса после пятого – ваше приложение будет обрабатывать первые пять клиентских запросов должным образом, а затем возвращать ошибку на каждом последующем запросе. Благодаря проверке живучести, когда это произойдет, оно должно быть перезапущено, позволяя ему снова правильно обрабатывать запросы клиентов.

Вы можете найти код нового приложения в архиве кода, прилагаемого к этой книге (в папке Chapter04/kubia-unhealthy). Я загрузил образ контейнера в хранилище Docker Hub, поэтому вам не нужно создавать его самостоятельно.

Вы создадите новый модуль, который включает в себя проверку живучести HTTP GET. В следующем ниже листинге показан YAML для модуля.

Листинг 4.1. Добавление проверки живучести в модуль: kubia-liveness-probe.yaml

```
apiVersion: v1
kind: pod
metadata:
  name: kubia-liveness
spec:
  containers:
  - image: luksa/kubia-unhealthy    <-- Этот образ содержит (несколько) нарушенное приложение
    name: kubia
    livenessProbe:                  <-- Проверка живучести, которая выполнит запрос HTTP GET
      httpGet:
        path: /                     <-- Путь к запросу в HTTP-запросе
        port: 8080                  <-- Сетевой порт, к которому проверка должна подключиться
```

Дескриптор модуля определяет проверку живучести `httpget`, которая сообщает Kubernetes периодически выполнять запросы HTTP GET на пути/порту `8080`, чтобы определять, является ли контейнер по-прежнему здоровым. Эти запросы начинаются сразу после запуска контейнера.

После пяти таких запросов (или реальных клиентских запросов) приложение начинает возвращать код состояния HTTP 500, который Kubernetes будет рассматривать как несработка проверки, и таким образом перезапустит контейнер.

4.1.3 Просмотр проверки живучести в действии

Для того чтобы увидеть работу проверки живучести, попробуйте сейчас создать модуль. Примерно через полторы минуты контейнер будет перезапущен. Вы сможете увидеть это, выполнив команду `kubectl get`:

```
$ kubectl get po kubia-liveness
NAME            READY  STATUS   RESTARTS  AGE
kubia-liveness  1/1    Running  1         2m
```

В столбце перезапусков `RESTARTS` показано, что контейнер модуля был перезапущен один раз (если подождать еще полторы минуты, он снова перезапускается, а затем цикл продолжится бесконечно).

Получение лога приложения аварийного контейнера

В предыдущей главе вы узнали, как распечатывать журнал приложения с помощью команды `kubectl logs`. Если ваш контейнер перезапускается, то команда `kubectl logs` будет показывать отчет в текущем контейнере.
Если вы хотите выяснить, почему предыдущий контейнер прекратил работу, вам потребуется увидеть эти журналы вместо журналов текущего контейнера. Это можно сделать с помощью параметра `--previous`:

```
$ kubectl logs mypod --previous
```

Как показано в следующем далее листинге, вы можете увидеть, почему контейнер нужно было перезапустить, посмотрев на распечатку команды `kubectl describe`.

Листинг 4.2. Описание модуля после перезапуска его контейнера

```
$ kubectl describe po kubia-liveness
Name:           kubia-liveness
...
Containers:
  kubia:
    Container ID:       docker://480986f8
    Image:              luksa/kubia-unhealthy
    Image ID:           docker://sha256:2b208508
    Port:
    State:              Running
      Started:          Sun, 14 May 2017 11:41:40 +0200
    Last State:         Terminated
      Reason:           Error
      Exit Code:        137
      Started:          Mon, 01 Jan 0001 00:00:00 +0000
      Finished:         Sun, 14 May 2017 11:41:38 +0200
    Ready:              True
    Restart Count:      1
    Liveness:           http-get http://:8080/ delay=0s timeout=1s
                        period=10s #success=1 #failure=3
    ...
Events:
... Killing container with id docker://95246981:pod "kubia-liveness ..."
    container "kubia" is unhealthy, it will be killed and re-created.
```

Контейнер в настоящий момент работает

Предыдущий контейнер завершил работу с ошибкой и вышел с кодом 137

Контейнер был перезапущен снова

Вы можете видеть, что контейнер в настоящее время работает, но он ранее завершил работу из-за ошибки. Код выхода был 137, что имеет особое значение – он обозначает, что процесс был завершен по внешнему сигналу. Число

137 – это сумма двух чисел: 128 + x, где x – номер сигнала, отправленного процессу, который вызвал его завершение. В этом примере x равняется 9, число сигнала SIGKILL, т. е. процесс был убит принудительно.

События, перечисленные в нижней части, показывают, почему контейнер был убит – Kubernetes обнаружил, что контейнер был нездоровым, поэтому он убил и воссоздал его снова.

> **ПРИМЕЧАНИЕ.** Когда контейнер убит, создается совершенно новый контейнер – это не тот же самый контейнер, который снова перезапускается.

4.1.4 Настройка дополнительных свойств проверки живучести

Возможно, вы заметили, что команда kubectl describe также отображает дополнительную информацию о проверке живучести:

```
Liveness: http-get http://:8080/ delay=0s timeout=1s period=10s #success=1
        ➥ #failure=3
```

Помимо параметров проверки живучести, которые вы указали явно, вы также можете увидеть дополнительные свойства, такие как delay, timeout, period и т. д. Фрагмент delay=0s показывает, что проверки начинаются сразу после запуска контейнера. Тайм-аут timeout установлен только на 1 секунду, поэтому контейнер должен вернуть отклик через 1 секунду, или проверка будет считаться несработавшей. Контейнер опробируется каждые 10 секунд (period=10s), и контейнер перезапускается после несработки проверки три раза подряд (#failure=3).

Эти дополнительные параметры можно индивидуально настроить во время определения проверки. Например, чтобы задать начальную задержку, добавьте в проверку живучести свойство initialDelaySeconds, как показано в следующем ниже листинге.

Листинг 4.3. Проверка живучести с первоначальной задержкой: kubia-liveness-probe-initial-delay.yaml

```
livenessProbe:
  httpGet:
    path: /
    port: 8080
  initialDelaySeconds: 15    <-- Перед исполнением первой проверки Kubernetes будет ждать 15 секунд
```

Если вы не зададите первоначальную задержку, проверка начнет опробирование контейнера сразу же после его запуска, что обычно приводит к несработке проверки, так как приложение не готово к получению запросов. Если количество несработок превышает пороговое значение несработки, контейнер

перезапускается, прежде чем он даже сможет начать откликаться на запросы должным образом.

> **СОВЕТ.** Всегда помните, что следует устанавливать первоначальную задержку с учетом времени запуска вашего приложения.

Я встречал такую ситуацию довольно часто, и пользователи были в замешательстве, почему их контейнер перезапускался. Но если бы они применили `kubectl describe`, то они бы увидели, что контейнер завершается кодом выхода 137 или 143, сообщая им, что работа модуля была завершена извне. Кроме того, список событий модуля показал бы, что контейнер был убит из-за неуспешной проверки живучести. Если вы видите, что это происходит при запуске модуля, то это потому, что вам не удалось установить `initialDelaySeconds` как надо.

> **ПРИМЕЧАНИЕ.** Код выхода 137 сигнализирует о том, что процесс был убит внешним сигналом (код выхода 128 + 9 (SIGKILL). Аналогичным образом код выхода 143 соответствует 128 + 15 (SIGTERM).

4.1.5 Создание эффективных проверок живучести

В случае модулей, работающих в рабочем окружении, вы всегда должны назначать проверку живучести. Без нее Kubernetes не имеет никакого способа узнать, живое ваше приложение или нет. До тех пор, пока процесс по-прежнему выполняется, Kubernetes будет считать контейнер здоровым.

Что должна проверять проверка живучести

Ваша упрощенная проверка живучести просто проверяет, откликается сервер или нет. Хотя она может показаться слишком простой, но даже такая проверка живучести делает чудеса, потому что она приводит к перезапуску контейнера, если работающий в контейнере веб-сервер перестает откликаться на запросы HTTP. В отличие от ситуации, когда проверка живучести отсутствует, это серьезное улучшение, и может быть достаточным в большинстве случаев.

Но для более серьезной проверки живучести вы настроите проверку для выполнения запросов по определенному URL-пути (например, `/health`) и заставите приложение выполнять внутреннюю проверку состояния всех жизненно важных компонентов, работающих внутри приложения, чтобы убедиться, что ни один из них не умер или не откликается.

> **СОВЕТ.** Убедитесь, что конечная точка HTTP `/health` не требует аутентификации; в противном случае опробирование всегда будет завершаться несработкой, что приведет к бесконечному перезапуску контейнера.

Обязательно проверяйте только внутренние части приложения и ничего из того, что зависит от внешнего фактора. Например, проверка живучести фронтенд-сервера не должна возвращать несработку, когда сервер не может подключиться к внутренней базе данных. Если основная причина находится в самой базе данных, перезапуск контейнера веб-сервера проблему не устранит.

Поскольку проверка живучести снова завершится несработкой, вы окажетесь в ситуации, когда контейнер будет перезапускаться повторно до тех пор, пока база данных не станет снова доступной.

Поддержание проверок легковесными

Проверки живучести не должны использовать слишком много вычислительных ресурсов и не должны занимать слишком много времени. По умолчанию опробирования исполняются относительно часто, и на их завершение допускается не больше одной секунды. Проверка, которая выполняет тяжелую работу, может значительно замедлить ваш контейнер. Позже в книге вы также узнаете о том, как ограничивать процессорное время, доступное для контейнера. Процессорное время проверки подсчитывается, исходя из квоты процессорного времени контейнера, поэтому наличие тяжеловесной проверки уменьшит процессорное время, доступное для главных процессов приложения.

СОВЕТ. Если вы запускаете приложение Java в контейнере, обязательно вместо проверки Exec, в которой, для того чтобы получить информацию о живучести, вы запускаете совершенно новую JVM, используйте проверку живучести HTTP GET. То же самое относится к любым приложениям на основе JVM или подобным приложениям, процедура запуска которых требует значительных вычислительных ресурсов.

Не утруждайтесь реализацией циклов в своих проверках

Вы уже видели, что порог несработки для проверки конфигурируем, и обычно, прежде чем контейнер будет убит, проверка должна показать несработку многократно. Но даже если вы установите порог несработки равным 1, Kubernetes повторит проверку несколько раз, прежде чем рассматривать его как одну неуспешную попытку. Поэтому внедрение собственного цикла в проверку является потраченным впустую усилием.

Подведение итогов относительно проверки живучести

Теперь вы понимаете, что Kubernetes поддерживает ваши контейнеры в рабочем состоянии путем их перезапуска, в случае если они терпят аварию либо если их проверки живучести не срабатывают. Это задание выполняется агентом Kubelet на узле, на котором размещен модуль, – компоненты плоскости управления Kubernetes, работающие на ведущем (ведущих) узле(ах), в этом процессе не участвуют.

Но если сам узел выходит из строя, то именно плоскость управления должна создать замены для всех модулей, которые упали вместе с узлом. Она не делает этого для модулей, которые вы создаете непосредственно. Эти модули не управляются ничем, кроме как службой Kubelet, а поскольку Kubelet работает на самом узле, она не может сделать ничего, если узел аварийно завершает работу.

Для того чтобы убедиться, что приложение перезапущено на другом узле, необходимо иметь модуль, управляемый контроллером репликации или аналогичным механизмом, который мы обсудим в остальной части этой главы.

4.2 Знакомство с контроллерами репликации

Контроллер репликации (ReplicationController) – это ресурс Kubernetes, который обеспечивает поддержание постоянной работы его модулей. Если модуль исчезает по любой причине, например в случае исчезновения узла из кластера или потому, что модуль был вытеснен из узла, контроллер репликации замечает отсутствующий модуль и создает сменный модуль.

Рисунок 4.1 показывает, что происходит, когда узел падает и уносит с собой два модуля. Модуль A был создан напрямую, и поэтому он является неуправляемым, в то время как модуль B управляется контроллером репликации. После аварийного завершения работы узла контроллер репликации создает новый модуль (модуль B2) для замены отсутствующего модуля B, тогда как модуль A теряется полностью – ничто его не воссоздаст заново.

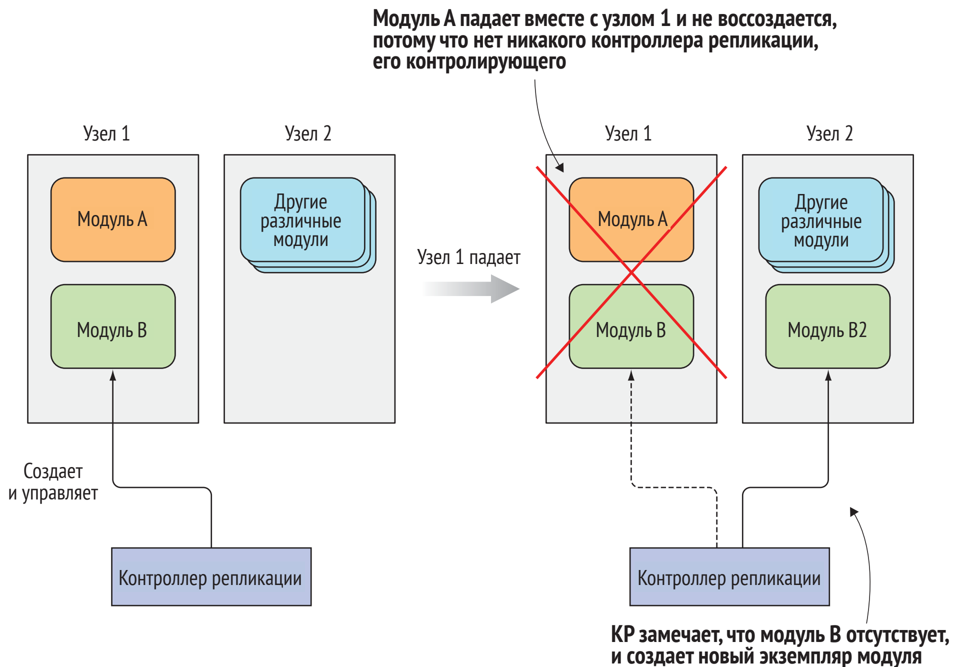


Рис. 4.1. Во время аварийного завершения работы узла воссоздаются только те части, которые поддерживаются контроллером репликации

Контроллер репликации на рисунке управляет только одним модулем, но контроллеры репликации, как правило, предназначены для создания и управления несколькими копиями (репликами) модуля. Отсюда контроллеры репликаций и получили свое название.

4.2.1 Работа контроллера репликации

Контроллер репликации постоянно отслеживает список запущенных модулей и удостоверяется, что фактическое количество модулей определенного «типа» всегда совпадает с требуемым числом. Если запущено слишком мало таких модулей, то из шаблона модуля создаются новые реплики. Если запущено слишком много таких модулей, то он удаляет лишние реплики.

Вам может быть интересно, как это может быть больше, чем нужное количество реплик. Это может произойти по нескольким причинам:

- кто-то создает модуль того же типа вручную;
- кто-то изменяет «тип» существующего модуля;
- кто-то уменьшает требуемое количество модулей и т. д.

Я использовал термин «тип» модуля несколько раз. Но такого не существует. Контроллеры репликации работают не на типах модулей, а на наборах модулей, которые соответствуют определенному селектору меток (вы узнали о них в предыдущей главе).

Знакомство с циклом контроллера по приведению в соответствие

Задача контроллера репликации состоит в том, чтобы убедиться, что точное количество модулей всегда совпадает с его селектором меток. Если это не так, то контроллер репликации выполняет соответствующее действие для приведения в соответствие фактического количества с требуемым. Работа контроллера репликации показана на рис. 4.2.

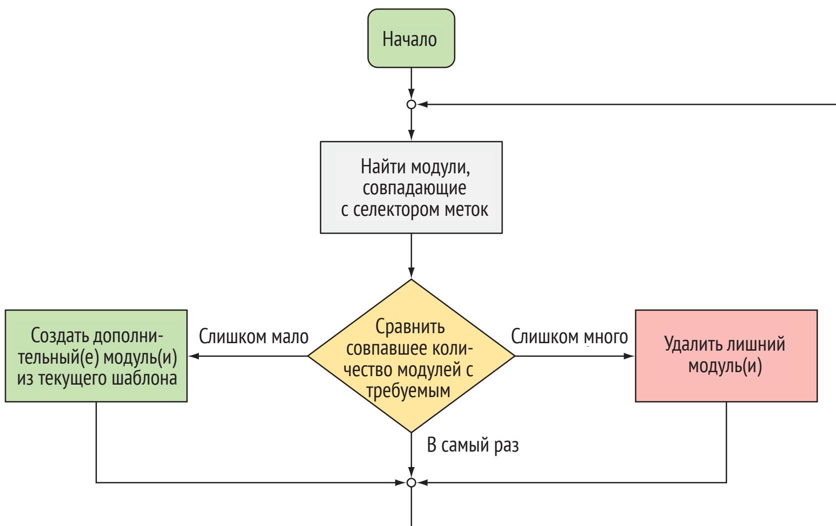


Рис. 4.2. Цикл контроллера репликации по приведению в соответствие

Три части контроллера репликации

Контроллер репликации состоит из трех основных частей (также показано на рис. 4.3):

- *селектор меток*, определяющий, какие модули находятся в области действия контроллера репликации;
- *количество реплик*, указывающее на требуемое количество модулей, которые должны быть запущены;
- *шаблон модуля*, используемый при создании новых реплик модуля.

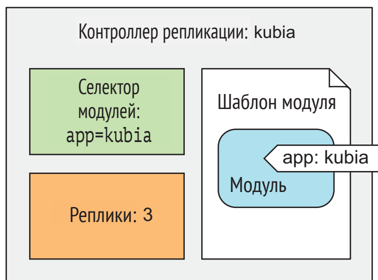


Рис. 4.3. Три ключевые части контроллера репликации (селектор модулей, количество реплик и шаблон модуля)

Количество реплик контроллера репликации, селектор меток и даже шаблон модуля могут быть изменены в любое время, но только изменения количества реплик влияют на то, как ведется работа с существующими модулями.

Эффект изменения селектора меток или шаблона модуля контроллера репликации

Изменения в селекторе меток и шаблоне модуля не влияют на существующие модули. Изменение селектора меток приводит к выпадению существующих модулей из области действия контроллера репликации, поэтому контроллер перестает о них заботиться. Контроллеры репликации также не заботятся о фактическом «содержимом» своих модулей (образах контейнеров, переменных среды и других аспеках) после создания модуля. Таким образом, шаблон влияет только на новые модули, создаваемые этим контроллером репликации. Эту процедуру можно представить как форму для печенья для вырезания новых модулей.

Преимущества использования контроллера репликации

Как и многое в Kubernetes, контроллер репликации, хотя и невероятно простая концепция, предоставляет или активирует следующие мощные функциональные средства:

- он гарантирует, что модуль (или несколько реплик модуля) всегда работает путем запуска нового модуля, когда существующий пропадает;
- когда узел кластера аварийно завершает работу, он создает сменные реплики для всех модулей, которые работали на отказавшем узле (которые находились под управлением контроллера репликации);

- обеспечивает простое горизонтальное масштабирование модулей – как ручное, так и автоматическое (см. раздел «Горизонтальное масштабирование модулей» в главе 15).

ПРИМЕЧАНИЕ. Экземпляр модуля никогда не переносится на другой узел. Вместо этого контроллер репликации создает совершенно новый экземпляр модуля, который не имеет отношения к экземпляру, который он заменяет.

4.2.2 Создание контроллера репликации

Давайте посмотрим, как создать контроллер репликации, а затем как он поддерживает ваши модули в работающем состоянии. Подобно модулям и другим ресурсам Kubernetes, вы создаете контроллер репликации, отправляя дескриптор JSON или YAML на сервер API Kubernetes.

Для вашего контроллера репликации вы создадите файл YAML под названием `kubia-rc.yaml`, как показано в следующем списке.

Листинг 4.4. Определение YAML контроллера репликации: kubia-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    app: kubia
template:
  metadata:
    labels:
      app: kubia
  spec:
    containers:
    - name: kubia
      image: luksa/kubia
      ports:
      - containerPort: 8080
```

Этот манифест определяет контроллер репликации (КР)

Имя этого контроллера репликации

Требуемое количество экземпляров модуля

Селектор модулей, определяющий, с какими модулями оперирует КР

Шаблон модуля для создания новых модулей

При отправке файла на сервер API Kubernetes создает новый контроллер репликации с именем `kubia`, который гарантирует, что три экземпляра модуля всегда совпадают с селектором меток `app=kubia`. Когда не хватает модулей, новые модули будут созданы из предоставленного шаблона модуля. Содержимое шаблона почти идентично определению модуля, созданному в предыдущей главе.

Метки модуля в шаблоне должны, безусловно, совпадать с селектором меток контроллера репликации; в противном случае контроллер будет созда-

вать новые модули бесконечно, потому что запуск нового модуля не приблизит фактическое количество реплик к требуемому количеству. Для того чтобы предотвратить такие сценарии, сервер API проверяет определение контроллера репликации и не принимает его, если оно неправильно сконфигурировано.

Не указывать селектор вообще – это тоже вариант. В этом случае он будет настроен автоматически из меток в шаблоне модуля.

> **СОВЕТ.** Не указывайте селектор модулей при определении контроллера репликации. Дайте системе Kubernetes извлекать его из шаблона модуля. Это сделает ваш YAML короче и проще.

Для того чтобы создать контроллер репликации, используйте команду `kubectl create`, которую вы уже знаете:

```
$ kubectl create -f kubia-rc.yaml
replicationcontroller "kubia" created
```

После создания контроллера репликации он начинает работать. Давайте посмотрим, что он делает.

4.2.3 Просмотр контроллера репликации в действии

Поскольку нет ни одного модуля с меткой `app=kubia`, контроллер репликации должен развернуть три новых модуля из шаблона модуля. Выведем список модулей, чтобы увидеть, сделал ли контроллер репликации то, что он должен:

```
$ kubectl get pods
NAME         READY   STATUS              RESTARTS   AGE
kubia-53thy  0/1     ContainerCreating   0          2s
kubia-k0xz6  0/1     ContainerCreating   0          2s
kubia-q3vkg  0/1     ContainerCreating   0          2s
```

И действительно! Требовалось три модуля, и он создал три модуля. Теперь он управляет этими тремя модулями. Давайте теперь немного с ними поработаем, чтобы увидеть, как контроллер репликации отвечает на эти действия.

Реакция контроллера репликации на удаление модуля

Сначала вы удалите один из модулей вручную, чтобы увидеть, как контроллер репликации немедленно запускает новый, в результате чего количество соответствующих модулей вернется к трем:

```
$ kubectl delete pod kubia-53thy
pod "kubia-53thy" deleted
```

При повторном выводе списка модулей отображаются четыре, так как удаленный модуль завершается, а новый модуль уже создан:

```
$ kubectl get pods
NAME          READY  STATUS             RESTARTS  AGE
kubia-53thy   1/1    Terminating        0         3m
kubia-oini2   0/1    ContainerCreating  0         2s
kubia-k0xz6   1/1    Running            0         3m
kubia-q3vkg   1/1    Running            0         3m
```

Контроллер репликации сделал свою работу снова. Очень милый маленький помощник, не так ли?

Получение информации о контроллере репликации

Теперь давайте посмотрим, какую информацию команда `kubectl get` показывает для контроллеров репликации:

```
$ kubectl get rc
NAME    DESIRED   CURRENT   READY   AGE
kubia   3         3         2       3m
```

> **ПРИМЕЧАНИЕ.** Мы используем `rc` как аббревиатуру для объекта `replicationcontroller`.

Вы видите три столбца, показывающие требуемое количество модулей, фактическое количество модулей и сколько из них готовы (вы узнаете, что это означает, в следующей главе, когда мы поговорим о проверках готовности).

Дополнительные сведения о контроллере репликации можно просмотреть с помощью команды `kubectl describe`, как показано в следующем ниже листинге.

Листинг 4.5. Отображение сведений о контроллере репликации с помощью команды kubectl describe

```
$ kubectl describe rc kubia
Name: kubia
Namespace:      default
Selector:       app=kubia
Labels:         app=kubia
Annotations:    <none>
Replicas:       3 current / 3 desired                    <-- Фактическое против требуемого количества экземпляров модуля
Pods Status:    4 Running / 0 Waiting / 0 Succeeded / 0 Failed   <-- Количество экземпляров модуля в расчете на статус модуля
Pod Template:
  Labels:       app=kubia
  Containers:   ...
  Volumes:      <none>
Events:                                                  <-- События, связанные с этим контроллером репликации
From                    Type    Reason           Message
----                    ------- ------           -------
```

```
replication-controller  Normal  SuccessfulCreate  Created pod: kubia-53thy
replication-controller  Normal  SuccessfulCreate  Created pod: kubia-k0xz6
replication-controller  Normal  SuccessfulCreate  Created pod: kubia-q3vkg
replication-controller  Normal  SuccessfulCreate  Created pod: kubia-oini2
```

Текущее количество реплик соответствует требуемому количеству, поскольку контроллер уже создал новый модуль. Он показывает четыре запущенных модуля, потому что завершающийся модуль по-прежнему считается запущенным, хотя он не учитывается в текущем количестве реплик.

Список событий внизу показывает действия, предпринятые контроллером репликации – на этот момент он создал четыре модуля.

Что именно заставило контроллер создать новый модуль

Контроллер откликается на удаление модуля путем создания нового сменного модуля (см. рис. 4.4). Дело в том, что в техническом плане он не откликается на само удаление, но результирующее состояние – несоответствующее количество модулей.

В то время как контроллер репликации немедленно уведомляется об удаляемом модуле (сервер API допускает, чтобы клиенты наблюдали за изменениями в ресурсах и списках ресурсов), это не то, что заставляет его создавать сменный модуль. Уведомление заставляет контроллер проверить фактическое количество модулей и предпринять соответствующее действие.

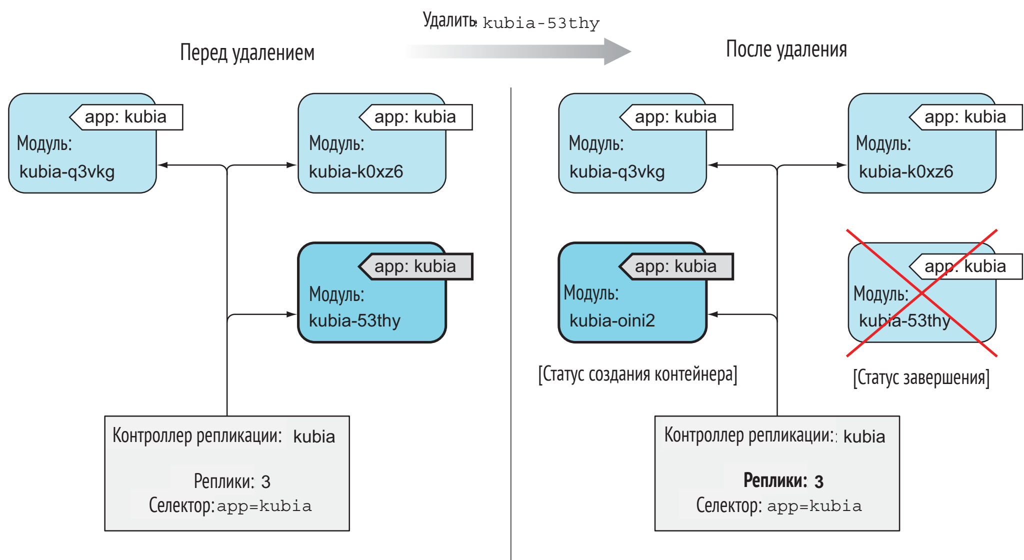


Рис. 4.4. Если модуль исчезает, контроллер репликации видит слишком мало модулей и создает новый сменный модуль

Отклик на аварийный сбой узла

Смотреть, как контроллер репликации откликается на ручное удаление модуля, не слишком интересно, поэтому давайте рассмотрим более подходящий пример. Если для выполнения этих примеров вы используете Google Kubernetes Engine, у вас есть кластер Kubernetes с тремя узлами. Сейчас вы отключите один из узлов от сети, чтобы симулировать аварийный сбой узла.

> **ПРИМЕЧАНИЕ.** Если вы используете Minikube, то выполнить это упражнение вы не сможете, поскольку у вас всего один узел, который действует и как ведущий, и как рабочий узел.

Если узел аварийно выходит из строя в мире, отличном от Kubernetes, системным администраторам потребуется вручную мигрировать приложения, работающие на этом узле, на другие машины. Kubernetes, однако, делает это автоматически. Вскоре после того, как контроллер репликации обнаружит, что его модули не работают, он развернет новые модули, чтобы их заменить.

Давайте посмотрим на эту процедуру в действии. Вам нужен доступ `ssh` в один из узлов с помощью команды `gcloud compute ssh`. Затем выключите его сетевой интерфейс с помощью команды `sudo ifconfig eth0`, как показано в следующем ниже листинге.

> **ПРИМЕЧАНИЕ.** Выберите узел, который выполняет, по крайней мере, один из ваших модулей, выведя список модулей с помощью параметра `-o wide`.

Листинг 4.6. Симулирование аварийного сбоя узла путем выключения его сетевого интерфейса

```
$ gcloud compute ssh gke-kubia-default-pool-b46381f1-zwko
Enter passphrase for key '/home/luksa/.ssh/google_compute_engine':
Welcome to Kubernetes v1.6.4!
...
luksa@gke-kubia-default-pool-b46381f1-zwko ~ $ sudo ifconfig eth0 down
```

Когда вы завершаете работу сетевого интерфейса, сеанс `ssh` перестанет отвечать на запросы, поэтому вам нужно открыть еще один терминал или выполнить жесткий выход из сеанса `ssh`. В новом терминале вы можете вывести список узлов, чтобы увидеть, обнаружил ли Kubernetes, что узел не работает. Это займет минуту или около того. Затем состояние узла отображается как `NotReady`:

```
$ kubectl get node
NAME                                   STATUS     AGE
gke-kubia-default-pool-b46381f1-opc5   Ready      5h
gke-kubia-default-pool-b46381f1-s8gj   Ready      5h
gke-kubia-default-pool-b46381f1-zwko   NotReady   5h      <- Узел не готов, потому что
                                                             он отсоединен от сети
```

Если вывести список модулей сейчас, то вы все равно увидите те же три модуля, что и раньше, потому что Kubernetes ожидает некоторое время, прежде чем перепланировать модули (в случае если узел недоступен из-за временного сбоя сети или потому что агент Kubelet перезапускается). Если узел остается недостижимым в течение нескольких минут, то статус модулей, которые были назначены этому узлу, изменяется на `Unknown`. В этот момент контроллер репликации немедленно развернет новый модуль. Вы можете увидеть это, снова выведя список модулей:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-oini2   1/1     Running   0          10m
kubia-k0xz6   1/1     Running   0          10m
kubia-q3vkg   1/1     Unknown   0          10m
kubia-dmdck   1/1     Running   0          5s
```


Глядя на возраст модулей, вы видите, что модуль `kubia-dmdck` – новый. У вас снова работают три экземпляра модуля, то есть контроллер репликации опять выполнил свою работу по приведению фактического состояния системы к требуемому состоянию.

То же самое происходит, если узел аварийно отказывает (или ломается, или становится недостижимым). Немедленное вмешательство человека не требуется. Система лечит себя автоматически.

Для того чтобы вернуть узел обратно, необходимо сбросить его с помощью следующей команды:

```
$ gcloud compute instances reset gke-kubia-default-pool-b46381f1-zwko
```

Когда узел снова загрузится, его статус должен вернуться в `Ready`, и модуль, статус которого был `Unknown`, будет удален.

4.2.4 Перемещение модулей в область и из области действия контроллера репликации

Модули, создаваемые контроллером репликации, не привязаны к контроллеру репликации каким-либо образом. В любой момент контроллер репликации управляет теми модулями, которые соответствуют его селектору меток. Путем изменения меток модуля он может быть удален или добавлен к области действия контроллера репликации. Его можно даже перемещать из одного контроллера репликации в другой.

> **СОВЕТ.** Хотя модуль не привязан к контроллеру репликации, модуль все же ссылается на него в поле `metadata.ownerReferences`, которое можно использовать, чтобы легко найти, к какому контроллеру репликации модуль принадлежит.

Если вы изменяете метки модуля, чтобы они больше не совпадали с селектором меток контроллера репликации, то модуль становится похожим на

любой другой созданный вручную модуль. Он больше ничем не управляется. Если узел модуля аварийно завершает свою работу, то этот модуль очевидным образом не переназначается. Но имейте в виду, что при изменении меток модуля контроллер репликации заметит, что один модуль отсутствует, и развернет новый модуль, чтобы его заменить.

Давайте попробуем с вашими модулями. Поскольку контроллер репликации управляет модулями с меткой `app=kubia`, необходимо либо удалить эту метку, либо изменить ее значение, чтобы переместить модуль из области действия контроллера репликации. Добавление другой метки не будет иметь никакого эффекта, потому что контроллер репликации не заботится тем, имеет модуль какие-либо дополнительные метки или нет. Он занимается только тем, имеет ли модуль все метки, на которые есть ссылки в селекторе меток.

Добавление меток в модули, управляемые контроллером репликации

Давайте подтвердим, что контроллер репликации не заботится, добавляете вы дополнительные метки к его управляемым модулям или нет:

```
$ kubectl label pod kubia-dmdck type=special
pod "kubia-dmdck" labeled

$ kubectl get pods --show-labels
NAME         READY  STATUS   RESTARTS  AGE  LABELS
kubia-oini2  1/1    Running  0         11m  app=kubia
kubia-k0xz6  1/1    Running  0         11m  app=kubia
kubia-dmdck  1/1    Running  0         1m   app=kubia,type=special
```

В один из модулей вы добавили специальную метку `type=special`. Вывод списка всех модулей снова показывает те же три модуля, как и раньше, потому что со стороны контроллера репликации никакого изменения не произошло.

Изменение меток управляемого модуля

Теперь вы измените метку `app=kubia` на что-то другое. Это приведет к тому, что модуль больше не будет совпадать с селектором меток контроллера репликации, оставив ему всего два совпадения с модулями. Поэтому контроллер репликации должен запустить новый модуль, чтобы вернуть количество к трем:

```
$ kubectl label pod kubia-dmdck app=foo --overwrite
pod "kubia-dmdck" labeled
```

Параметр `--overwrite` необходим; в противном случае `kubectl` распечатает только предупреждение и не изменит метку, чтобы предотвратить случайное изменение значения существующей метки при добавлении новой.

Вывод списка всех модулей снова должен теперь показать четыре модуля:

```
$ kubectl get pods -L app
NAME         READY  STATUS             RESTARTS  AGE  APP
```

```
kubia-2qneh  0/1    ContainerCreating  0         2s   kubia
kubia-oini2  1/1    Running            0         20m  kubia
kubia-k0xz6  1/1    Running            0         20m  kubia
kubia-dmdck  1/1    Running            0         10m  foo
```

Вновь созданный модуль заменяет модуль, который вы удалили из области действия контроллера репликации

Модуль больше не управляется контроллером репликации

ПРИМЕЧАНИЕ. Параметр `-L app` используется для показа метки app в столбце.

Теперь у вас в общей сложности четыре модуля: один – неуправляемый вашим контроллером репликации и три – управляемые. Среди них вновь созданный модуль.

На рис. 4.5 показано, что произошло при изменении меток модуля, чтобы они больше не соответствовали селектору модуля контроллера репликации. Вы можете видеть ваши три модуля и ваш контроллер репликации. После изменения метки модуля с `app=kubia` на `app=foo` контроллер репликации больше не заботится об этом модуле. Поскольку количество реплик контроллера установлено равным 3 и только два модуля соответствуют селектору меток, контроллер репликации запускает модуль `kubia-2qneh`, чтобы вернуть число к трем. Модуль `kubia-dmdck` теперь полностью независим и будет продолжать работать до тех пор, пока вы не удалите его вручную (вы можете сделать это сейчас, потому что он вам больше не нужен).

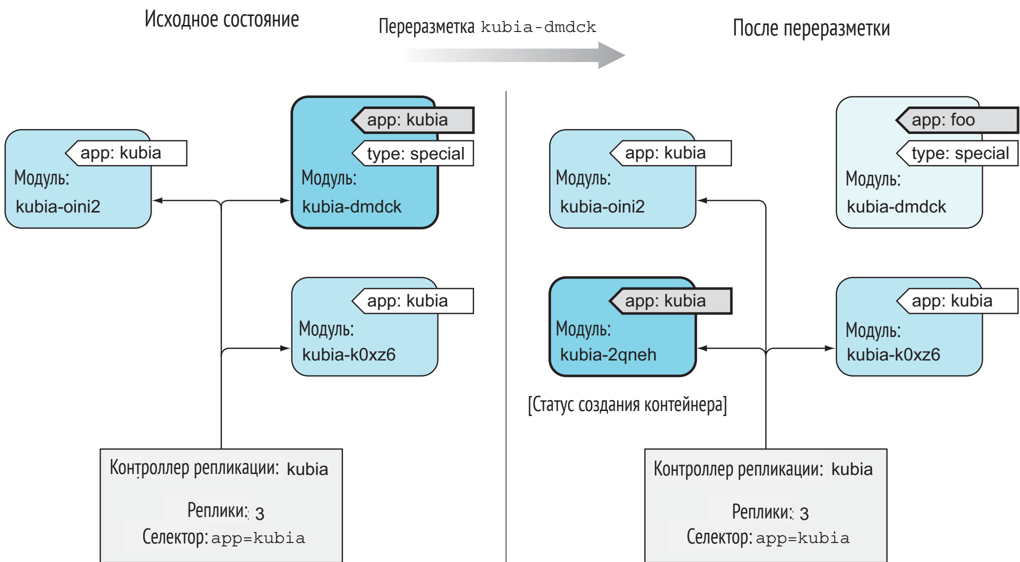


Рис. 4.5. Удаление модуля из области действия контроллера репликации путем изменения его меток

Удаление модулей из контроллеров на практике

Удалять модуль из области действия контроллера репликации бывает полезно, когда требуется выполнить действия на конкретном модуле. Например, у вас может быть ошибка, из-за которой ваш модуль начинает плохо себя вести после определенного периода времени или определенного события. Если вы знаете, что модуль неисправен, вы можете вывести его из области действия контроллера репликации, позволить контроллеру заменить его на новый, а затем заняться отладкой или поэкспериментировать с модулем любым способом по вашему усмотрению. Как только вы закончите, вы этот модуль удаляете.

Изменение селектора меток контроллера репликации

В качестве упражнения, чтобы убедиться, что вы полностью разбираетесь в контроллере репликации, что, по вашему мнению, произойдет, если вместо меток модуля вы изменили селектор меток контроллера репликации?

Если ваш ответ состоит в том, что это заставит все модули выпасть из области действия контроллера репликации, что приведет к созданию трех новых модулей, то вы абсолютно правы. И это показывает, что вы понимаете, как работают контроллеры репликации.

Kubernetes позволяет изменять селектор меток контроллера репликации, но это не относится к другим ресурсам, которые рассматриваются во второй половине этой главы и которые также используются для управления модулями. Вы никогда не будете изменять селектор меток контроллера, но вы будете регулярно менять его шаблон модуля. Давайте его рассмотрим.

4.2.5 Изменение шаблона модуля

Шаблон модуля контроллера репликации можно изменить в любое время. Изменение шаблона модуля подобно замене формы для печенья на другую. Это повлияет только на те печенья, которые вы будете вырезать позже, и не повлияет на те, которые вы уже вырезали (см. рис. 4.6). Для того чтобы изменить старые модули, необходимо удалить их и позволить контроллеру репликации заменить их новыми на основе нового шаблона.

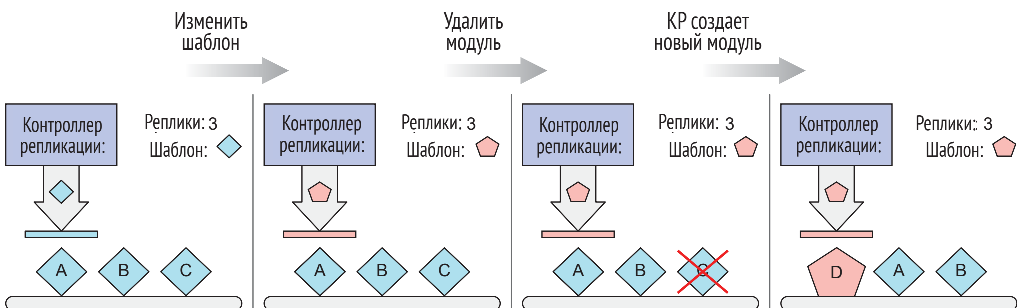


Рис. 4.6. Изменение шаблона модуля контроллера репликации влияет только на модули, создаваемые впоследствии, и не влияет на существующие модули

В качестве упражнения можно попробовать отредактировать контроллер репликации и добавить метку в шаблон модуля. Изменить контроллер репликации можно с помощью следующей ниже команды:

```
$ kubectl edit rc kubia
```

Она откроет определение YAML контроллера репликации в текстовом редакторе, установленном по умолчанию. Найдите секцию шаблона модуля (template) и добавьте новую метку в метаданные. После сохранения изменений и выхода из редактора `kubectl` обновит контроллер репликации и напечатает следующее ниже сообщение:

```
replicationcontroller "kubia" edited
```

Теперь вы можете вывести список модулей и их меток снова и подтвердить, что они не изменились. Но если вы удалите модули и дождетесь их замены, то увидите новую метку.

Такого рода редактирование контроллера репликации, чтобы изменить образ контейнера в шаблоне модуля, удаляя существующие модули и позволяя им быть замененными на новые из нового шаблона, может быть использовано для обновления модулей, но вы узнаете более оптимальный способ в главе 9.

Конфигурирование правки для kubectl с целью использования другого текстового редактора

Вы можете поручить `kubectl` использовать текстовый редактор по вашему выбору путем установки переменной окружения KUBE_EDITOR. Например, если для редактирования ресурсов Kubernetes вы хотите использовать nano, выполните следующую ниже команду (или поместите ее в файл `~/.bashrc` либо эквивалентный ему):

```
export KUBE_EDITOR="/usr/bin/nano"
```

Если переменная среды KUBE_EDITOR не задана, то `kubectl edit` откатит назад к использованию редактора, установленного по умолчанию, который обычно настраивается с помощью переменной среды EDITOR.

4.2.6 Горизонтальное масштабирование модулей

Вы видели, как контроллер репликации гарантирует, что всегда выполняется заданное количество экземпляров модуля. Поскольку изменить требуемое количество реплик невероятно просто, это также означает, что задача горизонтального масштабирования модулей становится тривиальной.

Масштабирование количества модулей вверх или вниз делается так же просто, как изменение значения поля реплик (replicas) в ресурсе контроллера репликации. После изменения контроллер репликации либо увидит слишком

много существующих модулей (при уменьшении масштаба) и удалит часть из них, либо увидит слишком мало (при увеличении масштаба) и создаст дополнительные модули.

Масштабирование контроллера репликации

Ваш контроллер репликации поддерживал непрерывную работу трех экземпляров модуля. Сейчас вы собираетесь промасштабировать это количество до 10. Как вы помните, вы уже масштабировали контроллер репликации в главе 2. Можно использовать ту же команду, что и раньше:

```
$ kubectl scale rc kubia --replicas=10
```

Но на этот раз все будет по-другому.

Масштабирование контроллера репликации путем изменения его определения

Вместо того чтобы использовать команду `kubectl scale`, вы будете масштабировать его декларативным способом, редактируя определение контроллера репликации:

```
$ kubectl edit rc kubia
```

Когда откроется текстовый редактор, найдите поле `spec.replicas` и поменяйте его значение на 10, как показано в следующем ниже листинге.

Листинг 4.7. Редактирование КР в текстовом редакторе с помощью команды kubectl edit

```
# Отредактируйте приведенный ниже объект. Строки, начинающиеся с '#',
# будут проигнорированы, а пустой файл прервет редактирование.
# Если при сохранении возникнет ошибка, этот файл будет открыт вновь
# с соответствующими несработками.
apiVersion: v1
kind: ReplicationController
metadata:
  ...
spec:
  replicas: 3        <-- Измените число 3 на число 10 в этой строке
  selector:
    app: kubia
  ...
```

Когда вы сохраните файл и закроете редактор, контроллер репликации обновится и немедленно промасштабирует количество модулей до 10:

```
$ kubectl get rc
NAME   DESIRED  CURRENT  READY  AGE
kubia  10       10       4      21m
```

Вот и все. Если команда `kubectl scale` выглядит так, как будто вы сообщаете Kubernetes, что нужно сделать, то теперь гораздо понятнее, что вы делаете декларативное изменение до требуемого состояния контроллера репликации и не сообщаете Kubernetes что-то делать.

Масштабирование вниз с помощью команды kubectl scale

Теперь промасштабируем вниз до 3. Для этого используется команда `kubectl scale`:

```
$ kubectl scale rc kubia --replicas=3
```

Эта команда лишь изменяет поле `spec.replicas` определения контроллера репликации – так же, как при его изменении с помощью команды `kubectl edit`.

Декларативный подход к масштабированию

Горизонтальное масштабирование модулей в Kubernetes – это вопрос вашего желания: «требуется *x* количество работающих экземпляров». Вы не говорите Kubernetes, что или как это сделать. Вы просто указываете требуемое состояние.

Этот декларативный подход упрощает взаимодействие с кластером Kubernetes. Представьте, что вам нужно вручную определить текущее количество работающих экземпляров, а затем явно указать Kubernetes, сколько дополнительных экземпляров нужно запустить. Такой подход требует больше работы и гораздо более подвержен ошибкам. Изменить простое число намного проще, и в главе 15 вы узнаете, что если включить горизонтальное автоматическое масштабирование модулей, то и это может быть сделано самим Kubernetes.

4.2.7 Удаление контроллера репликации

При удалении контроллера репликации с помощью команды `kubectl delete` модули также удаляются. Но поскольку модули, созданные контроллером репликации, не являются неотъемлемой частью контроллера репликации и лишь им управляются, можно удалить только контроллер репликации и оставить модули работающими, как показано на рис. 4.7.

Это может быть полезно, когда у вас изначально есть набор модулей, управляемых контроллером репликации, а затем вы принимаете решение заменить контроллер репликации, к примеру, на набор реплик (ReplicaSet) (вы узнаете о них далее). Это можно сделать, не затрагивая модулей и не прерывая их работу при замене контроллера репликации, который ими управляет.

При удалении контроллера репликации с помощью команды `kubectl delete` его модули можно сохранить, передав команде параметр `--cascade=false`. Попробуем прямо сейчас:

```
$ kubectl delete rc kubia --cascade=false
replicationcontroller "kubia" deleted
```

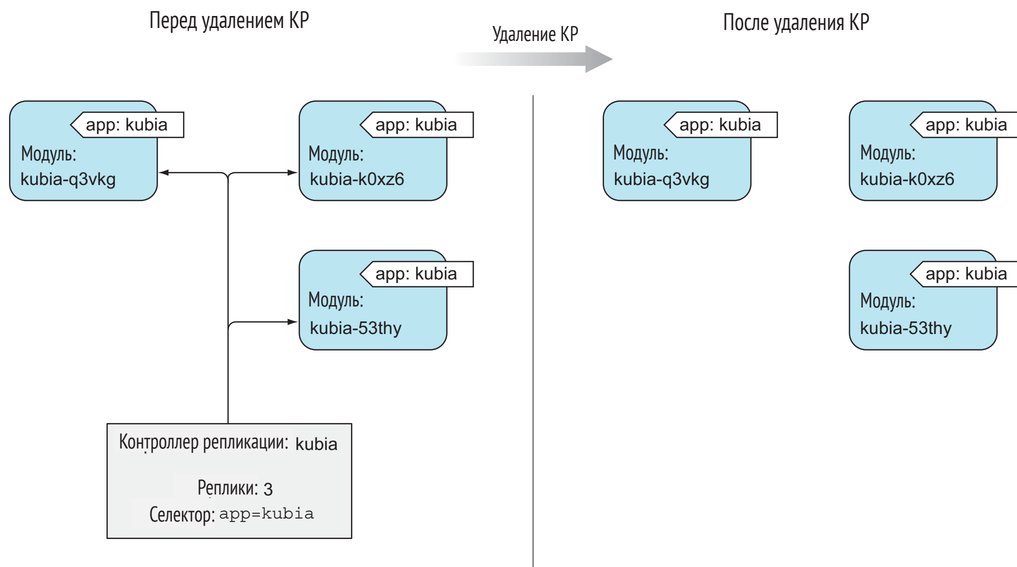


Рис. 4.7. Удаление контроллера репликации с параметром --cascade=false оставляет модули неуправляемыми

Вы удалили контроллер репликации, так что теперь модули существуют сами по себе. Они больше не управляются. Но вы всегда можете создать новый контроллер репликации с правильным селектором меток и снова сделать их управляемыми.

4.3 Использование набора реплик вместо контроллера репликации

Первоначально контроллеры репликации (ReplicationController) были единственным компонентом Kubernetes для репликации модулей и изменения их приписки при аварийном сбое узлов. Позже был введен аналогичный ресурс под названием набора реплик (ReplicaSet). Этот ресурс является новым поколением контроллера репликации и заменяет его полностью (контроллеры репликации в конечном итоге будут объявлены устаревшими).

Вы могли бы начать эту главу с создания набора реплик вместо контроллера репликации, но я посчитал, что было бы неплохо начать с того, что изначально имелось в Kubernetes. Кроме того, вы по-прежнему будете встречать использование контроллеров репликации в естественных условиях, так что осведомленность о них пойдет вам только на пользу. В связи с этим отныне вам всегда следует вместо контроллеров репликации создавать наборы реплик. Они почти идентичны, поэтому у вас не должно возникнуть никаких проблем с их использованием.

Обычно они создаются не напрямую, а автоматически, при создании ресурса более высокого уровня – развертывания, о чем вы узнаете в главе 9. В любом случае, вы должны понимать наборы реплик, поэтому давайте посмотрим, чем они отличаются от контроллеров репликации.

4.3.1 Сравнение набора реплик с контроллером репликации

Набор реплик ведет себя точно так же, как контроллер репликации, но имеет более выразительные селекторы модуля. В то время как селектор меток в контроллере репликации позволяет выбирать только те модули, которые включают в себя определенную метку, селектор набора реплик также позволяет выбирать модули, в которых недостает определенной метки, либо модули, которые содержат определенный ключ метки, независимо от его значения.

Кроме того, например, один контроллер репликации не может одновременно выбирать модули с меткой `env=production` и с меткой `env=devel`. Он может выбирать либо модули с меткой `env=production`, либо модули с меткой `env=devel`. Однако один набор реплик может выбирать оба набора модулей и рассматривать их как единую группу.

Точно так же контроллер репликации не может выбирать модули, основываясь только на присутствии ключа метки, независимо от его значения, тогда как набор меток может. Например, набор меток может выбирать все модули, которые включают метку с ключом `env`, независимо от его фактического значения (вы можете представить это как `env=*`).

4.3.2 Определение набора реплик

Сейчас вы создадите ресурс ReplicaSet, чтобы увидеть, как потерянные модули, созданные вашим контроллером репликации и оставленные ранее, теперь могут быть приняты набором реплик. Сначала вы перепишете свой контроллер репликации в набор реплик, создав новый файл под названием `kubia-replicaset.yaml` с содержимым в следующем ниже листинге.

Листинг 4.8. Определение набора реплик в YAML: kubia-replicaset.yaml

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubia
```

Наборы реплик не являются частью API v1, но принадлежат группе API apps и версии v1beta2 (к строкам apiVersion и kind)

Вы используете более простой селектор matchLabels, который очень похож на селектор контроллера репликации (к строкам matchLabels и app)

```
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
```

Шаблон такой же, как в контроллере репликации

В первую очередь нужно отметить, что наборы реплик не входят в v1 API, поэтому вы должны убедиться, что при создании ресурса вы задали правильную версию `apiVersion`. Вы создаете ресурс типа ReplicaSet, который имеет то же содержимое, что и ранее созданный контроллер репликации.

Разница только в селекторе. Вместо перечисления меток, которые модули должны иметь, прямо под свойством `selector` вы задаете их под `selector.matchLabels`. В наборе реплик это более простой (и менее выразительный, чем другие) способ определения селекторов меток. Позже вы также рассмотрите более выразительный вариант.

Об атрибуте версии API

Это ваша первая возможность увидеть, что свойство `apiVersion` определяет две вещи:

- группу API (в данном случае это приложения `apps`);
- актуальную версию API (`v1beta2`).

Вы увидите в книге, что некоторые ресурсы Kubernetes находятся в так называемой основной группе API, которую не нужно указывать в поле `apiVersion` (вы просто указываете версию – например, при определении ресурсов модулей вы использовали `apiVersion: v1`). Другие ресурсы, которые были представлены в более поздних версиях Kubernetes, подразделяются на несколько групп API. Обратитесь к сводной таблице на первом форзаце книги, чтобы увидеть все ресурсы и их соответствующие группы API.

Поскольку у вас по-прежнему имеется три модуля, соответствующих селектору `app=kubia`, запущенному ранее, создание этих реплик не приведет к созданию новых модулей. Набор реплик возьмет эти три модуля под свое крыло.

4.3.3 Создание и исследование набора реплик

С помощью команды `kubectl create` создадим набор реплик из файла YAML. После этого вы сможете изучить набор реплик с помощью команд `kubectl get` и `kubectl describe`:

```
$ kubectl get rs
NAME     DESIRED   CURRENT    READY    AGE
kubia    3         3          3        3s
```

СОВЕТ. Используйте аббревиатуру `rs`, которая обозначает `replicaset`.

```
$ kubectl describe rs
Name:           kubia
Namespace:      default
Selector:       app=kubia
Labels:         app=kubia
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=kubia
  Containers:   ...
  Volumes:      <none>
Events:         <none>
```

Как видите, набор реплик ничем не отличается от контроллера репликации. Он показывает, что у него есть три реплики, соответствующие селектору. Если вы выведете список всех модулей, то увидите, что это по-прежнему такие же 3 модуля, которые у вас были раньше. Набор реплик не создал новых.

4.3.4 Использование более выразительных селекторов меток набора реплик

Основные улучшения в наборе реплик, в отличие от контроллеров репликации, заключены в их более выразительных селекторах меток. В первом примере набора реплик вы преднамеренно использовали более простой селектор `matchLabels`, чтобы увидеть, что наборы реплик ничем не отличаются от контроллеров репликации. Теперь вы перепишете этот селектор, чтобы использовать более мощное свойство `matchExpressions`, как показано в следующем ниже листинге.

Листинг 4.9. Селектор matchExpressions: kubia-replicaset-matchexpressions.yaml

```
selector:
  matchExpressions:
    - key: app
      operator: In
      values:
        - kubia
```

Этот селектор требует, чтобы модуль содержал метку с ключом "app"

Значение метки должно быть "kubia"

ПРИМЕЧАНИЕ. Показан только селектор. Вы найдете полное определение реплик в архиве кода, прилагаемого к этой книге.

В селектор можно добавлять дополнительные выражения. Как и в примере, каждое выражение должно содержать ключ, оператор и, возможно (в зависимости от оператора), список значений. Вы увидите четыре допустимых оператора:

- `In` – значение метки должно совпадать с одним из указанных значений `values`;
- `NotIn` – значение метки не должно совпадать с любым из указанных значений `values`;
- `Exists` – модуль должен содержать метку с указанным ключом (значение не важно). При использовании этого оператора не следует указывать поле `values`;
- `DoesNotExist` – модуль не должен содержать метку с указанным ключом. Свойство `values` не должно быть указано.

Если вы указали несколько выражений, то, для того чтобы селектор совпал с модулем, все эти выражения должны оказываться истинными. Если указаны и `matchLabels`, и `matchExpressions`, то, чтобы модуль отождествился с селектором, все метки должны совпадать, а все выражения должны оказываться истинными.

4.3.5 Подведение итогов относительно наборов реплик

Это было краткое знакомство с наборами реплик (ReplicaSet) как альтернативой контроллерам репликации (ReplicationController). Помните, что всегда следует использовать их вместо контроллеров репликации, но в развертываниях других людей вы по-прежнему можете встречать контроллеры репликации.

Теперь удалите набор реплик, чтобы немного очистить кластер. Вы можете удалить набор реплик точно так же, как удаляете контроллер репликации:

```
$ kubectl delete rs kubia
replicaset "kubia" deleted
```

Удаление набора реплик должно удалить все модули. Выведите список модулей, чтобы подтвердить, что это так.

4.4 Запуск ровно одного модуля на каждом узле с помощью набора демонов (DaemonSet)

И контроллеры репликации (ReplicationController), и наборы реплик (ReplicaSet) используются для запуска определенного количества модулей, развернутых в любом месте кластера Kubernetes. Но существуют определенные случаи, когда требуется, чтобы модуль работал на каждом узле в кластере (и каждый узел должен выполнять ровно один экземпляр модуля, как показано на рис. 4.8).

Эти случаи включают модули, связанные с обеспечением работы инфраструктуры, выполняющие операции системного уровня. Например, требуется на каждом узле запустить сборщик логов и монитор ресурсов. Еще одним хорошим примером является собственный процесс kube-proxy системы Kubernetes, который должен работать на всех узлах, чтобы заставлять службы работать.

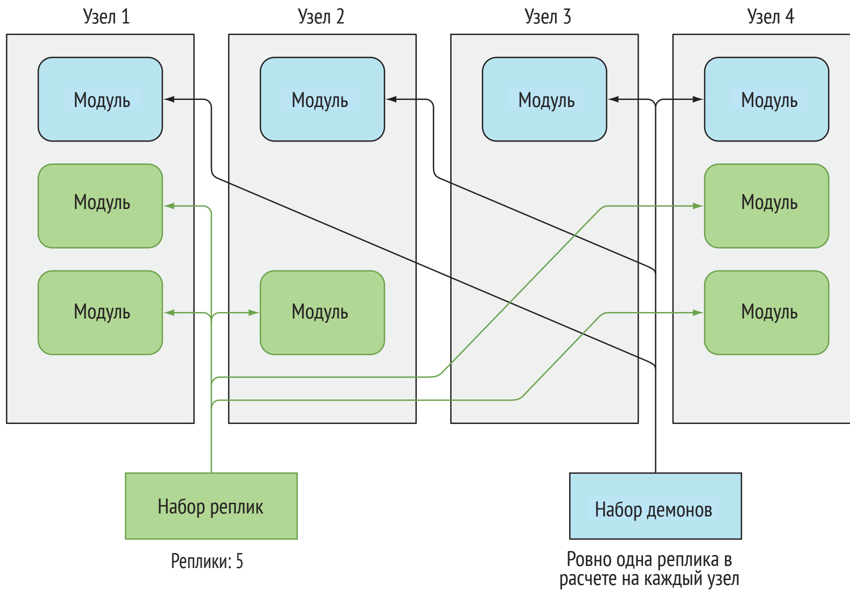


Рис. 4.8. Наборы демонов запускают только одну реплику модуля на каждом узле, в то время как наборы реплик разбрасывают их по всему кластеру случайным образом

За пределами Kubernetes такие процессы обычно запускаются с init-скриптов или демона `systemd` во время начальной загрузки узла. На узлах Kubernetes вы по-прежнему можете использовать `systemd` для запуска системных процессов, но тогда вы не сможете воспользоваться всеми возможностями, которые предоставляет Kubernetes.

4.4.1 Использование набора демонов для запуска модуля на каждом узле

Для запуска модуля на всех узлах кластера создается объект DaemonSet, который похож на объекты ReplicationController или ReplicaSet, за исключением того, что модули, созданные набором демонов, уже имеют заданный целевой узел и пропускают планировщик Kubernetes. Они не разбросаны по кластеру случайным образом.

Набор демонов гарантированно создает столько модулей, сколько имеется узлов, и разворачивает каждый на своем узле, как показано на рис. 4.8.

В отличие от набора реплик (или контроллера репликации), который гарантирует, что требуемое количество реплик модуля существует в кластере,

объект набор демонов не имеет никакого понятия о требуемом количестве реплик. Это и не нужно, потому что его задача состоит в том, чтобы гарантировать, что модуль, соответствующий его селектору, работает на каждом узле.

Если узел падает, то это не заставляет объект DaemonSet создавать модуль в другом месте. Но когда новый узел добавляется в кластер, набор демонов немедленно развертывает в нем новый экземпляр модуля. Он также делает то же самое, если кто-то случайно удаляет один из модулей, оставляя узел без модуля объекта DaemonSet. Как и набор реплик, набор демонов создает модуль из сконфигурированного в нем шаблона модуля.

4.4.2 Использование набора демонов для запуска модуля только на определенных узлах

Набор демонов развертывает модули на всех узлах кластера, если не указано, что модули должны выполняться только на подмножестве всех узлов. Это делается путем задания свойства `nodeSelector` в шаблоне модуля, который является частью определения ресурса DaemonSet (аналогично шаблону модуля в определении ресурса ReplicaSet или ReplicationController).

Вы уже использовали селекторы узлов для развертывания модуля на определенных узлах в главе 3. Селектор узлов в наборе демонов похож – он определяет узлы, на которых набор демонов должен развернуть свои модули.

> **ПРИМЕЧАНИЕ.** Позже в этой книге вы узнаете, что узлы могут быть сделаны неназначаемыми (unschedulable), предотвращая развертывание на них модулей. DaemonSet будет развертывать модули даже на таких узлах, поскольку атрибут unschedulable используется только планировщиком Scheduler, который применяется для размещения модулей, созданных через ReplicaSet и ReplicationController, тогда как модули, управляемые DaemonSet, полностью обходят планировщик. Это, как правило, желательно, потому что наборы демонов предназначены для запуска системных служб, которые обычно должны работать даже на неназначаемых узлах.

Объяснение наборов демонов на примере

Представим себе демон под названием `ssd-monitor`, который должен работать на всех узлах, содержащих твердотельный накопитель (SSD). Вы создадите объект DaemonSet, который запускает этот демон на всех узлах, помеченных как имеющие SSD. Администраторы кластера добавили метку `disk=ssd` во все такие узлы, поэтому вы создадите объект DaemonSet с помощью селектора узлов, который выбирает только узлы с этой меткой, как показано на рис. 4.9.

Создание YAML-определения ресурса DaemonSet

Вы создадите ресурс DaemonSet, который запускает фиктивный процесс `ssd-monitor`, печатающий «SSD OK» на стандартный вывод каждые пять секунд. Я уже подготовил образ фиктивного контейнера и отправил его в хра-

нилище Docker Hub, чтобы вы могли использовать его вместо создания собственного. Создайте YAML для ресурса DaemonSet, как показано в следующем ниже листинге.

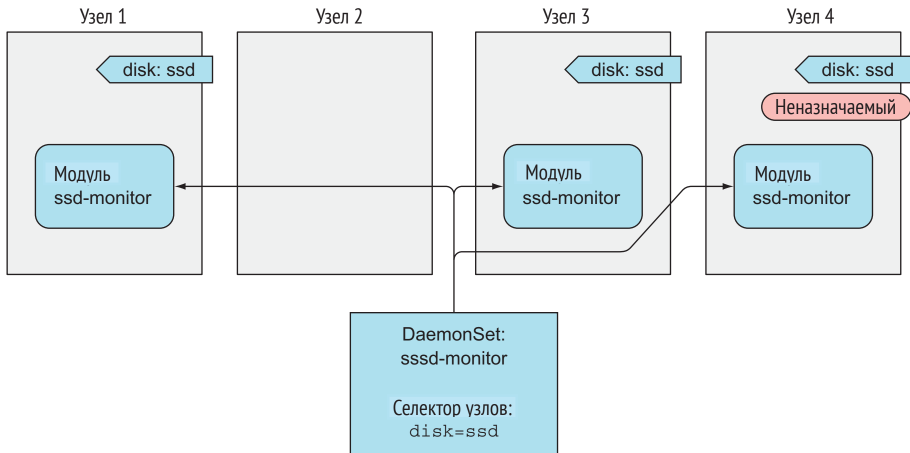


Рис. 4.9. Использование объекта DaemonSet с селектором узлов для развертывания системных модулей только на определенных узлах

Листинг 4.10. YAML для объекта DaemonSet: ssd-monitor-daemonset.yaml

```
apiVersion: apps/v1beta2        ← Объекты DaemonSet находятся
kind: DaemonSet                   в группе apps API версии v1beta2
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:             ← Шаблон модуля включает в себя селектор узлов,
        disk: ssd                 который отбирает узлы с меткой disk=ssd
      containers:
      - name: main
        image: luksa/ssd-monitor
```

Вы определяете объект DaemonSet, который будет запускать модуль с одним контейнером на основе образа контейнера `luksa/ssd-monitor`. Экземпляр этого модуля будет создан для каждого узла, имеющего метку `disk=ssd`.

Создание набора демонов

Вы создадите объект DaemonSet так же, как вы всегда создаете ресурсы из файла YAML:

```
$ kubectl create -f ssd-monitor-daemonset.yaml
daemonset "ssd-monitor" created
```

Давайте взглянем на созданный набор демонов:

```
$ kubectl get ds
NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE-SELECTOR
ssd-monitor  0         0         0       0            0           disk=ssd
```

Эти нули выглядят странно. Разве набор демонов не развернул никаких модулей? Выведем список модулей:

```
$ kubectl get po
No resources found.
```

Где же модули? Вы поняли, что происходит? Да, совершенно верно, вы забыли пометить узлы меткой `disk=ssd`. Нет проблем – вы можете сделать это сейчас. Набор демонов должен обнаружить, что метки узлов изменились, и развернуть узел на всех узлах с соответствующей меткой. Посмотрим, правда ли это.

Добавление необходимой метки в узел(ы)

Независимо от того, используете вы Minikube, GKE или другой многоузловой кластер, вам нужно сначала вывести список узлов, потому что при нанесении на узел метки вам нужно знать его имя:

```
$ kubectl get node
NAME       STATUS   AGE   VERSION
Minikube   Ready    4d    v1.6.0
```

Теперь на один из ваших узлов добавьте метку `disk=ssd`, как тут:

```
$ kubectl label node minikube disk=ssd
node "minikube" labeled
```

> **ПРИМЕЧАНИЕ.** Замените `minikube` на имя одного из ваших узлов, если вы не используете Minikube.

Теперь набор демонов должен был создать один модуль. Давайте посмотрим:

```
$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
ssd-monitor-hgxwq   1/1     Running   0          35s
```

Так, пока все хорошо. Если у вас есть несколько узлов и вы добавляете одну и ту же метку в последующие узлы, то вы увидите, что DaemonSet разворачивает модули для каждого из них.

Удаление необходимой метки из узла

Теперь представьте, что вы допустили ошибку и неправильно обозначили один из узлов. Он имеет вращающийся диск, а не SSD. Что произойдет, если изменить метку узла?

```
$ kubectl label node minikube disk=hdd --overwrite
node "minikube" labeled
```

Давайте посмотрим, имеет ли изменение какое-либо влияние на модуль, который работал на том узле:

```
$ kubectl get po
NAME                READY   STATUS        RESTARTS   AGE
ssd-monitor-hgxwq   1/1     Terminating   0          4m
```

Работа модуля завершается. Но вы знали, что это произойдет, верно? Это завершает ваше исследование наборов деменов (DaemonSet), поэтому вы можете удалить ваш объект набор демонов `ssd-monitor`. Если у вас по-прежнему работают другие модули демона, вы увидите, что удаление объекта DaemonSet также удаляет эти модули.

4.5 Запуск модулей, выполняющих одну заканчиваемую задачу

До сих пор мы говорили только о модулях, которым нужно работать постоянно. У вас будут иметься случаи, где вам требуется запустить задачу, которая завершается после окончания своей работы. Объекты ReplicationController, ReplicaSet и DaemonSet выполняют непрерывные задачи, которые не считаются заканчиваемыми. Процессы в таких модулях перезапускаются, когда они прекращаются. Но в заканчиваемой задаче, после завершения процесса, его не следует перезапускать снова.

4.5.1 Знакомство с ресурсом Job

Kubernetes включает поддержку такого функционала посредством ресурса Job, похожего на другие ресурсы, которые мы обсуждали в этой главе, но он позволяет запускать модуль, контейнер которого не перезапускается, когда процесс, запущенный внутри, заканчивается успешно. После этого модуль считается завершенным.

В случае аварийного завершения работы узла модули на этом узле, управляемые заданием (Job), будут переназначены на другие узлы, как и модули

набора реплик (ReplicaSet). В случае сбоя самого процесса (когда процесс возвращает код выхода с ошибкой) задание может быть настроено либо на перезапуск контейнера, либо нет.

На рис. 4.10 показано, как модуль, созданный заданием, переназначается на новый узел, если узел, на который он был первоначально назначен, завершает работу аварийно. На рисунке также показаны неуправляемый модуль, который не переназначается, и модуль, поддерживаемый набором реплик, который переназначается.

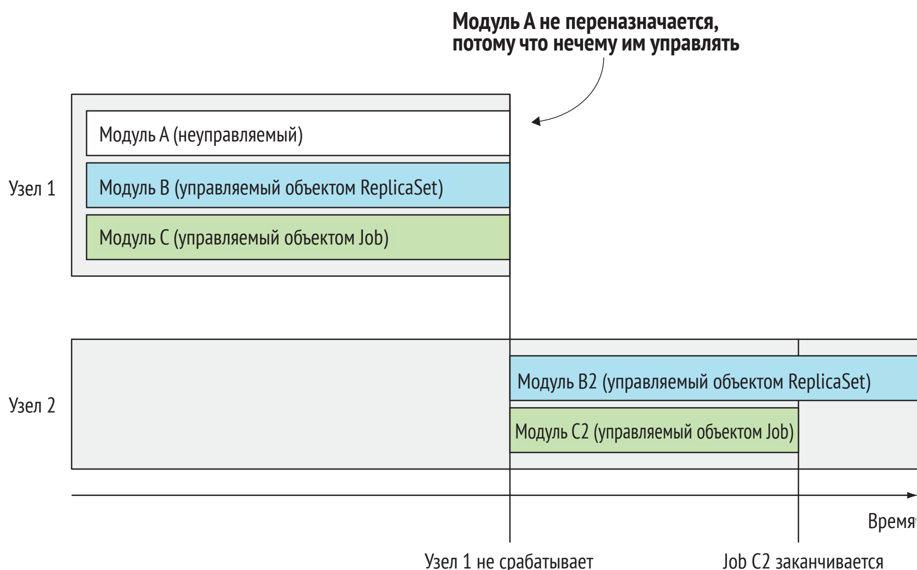


Рис. 4.10. Модули, управляемые заданиями, переназначаются до тех пор, пока задания не завершатся успешно

Например, задания (Job) полезны для нерегулярных задач, где крайне важно, чтобы задача закончилась правильно. Можно запустить задачу в неуправляемом модуле и дождаться ее окончания, но в случае аварийного сбоя узла или вытеснения модуля из узла во время выполнения задачи потребуется вручную создать ее заново. Делать это вручную не имеет смысла, в особенности если работа занимает несколько часов.

Примером такого задания может служить хранение данных в каком-либо месте, а также их преобразование и экспорт. Вы собираетесь эмулировать это, запустив образ контейнера, собранный поверх образа `busybox`, который вызывает команду `sleep` в течение двух минут. Я уже собрал образ и отправил его в хранилище Docker Hub, но вы можете заглянуть в его Dockerfile в архиве кода, прилагаемого к этой книге.

4.5.2 Определение ресурса Job

Создайте манифест ресурса Job, как показано в следующем ниже листинге.

Листинг 4.11. Определение ресурса Job в файле YAML: exporter.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      restartPolicy: OnFailure
      containers:
      - name: main
        image: luksa/batch-job
```

Объекты Job в группе API batch версии v1

Если вы не указываете селектор модуля (он будет создан на основе меток в шаблоне модуля)

Объекты Job не могут использовать принятую по умолчанию политику перезапуска, т. е. Always

Задания являются частью группы API `batch` и версии API `v1`. YAML определяет ресурс с типом Job, который будет выполнять образ `luksa/batch-job`. Этот образ вызывает процесс, который выполняется ровно 120 секунд, а затем заканчивается.

В секции спецификации модуля можно указать, что следует делать системе Kubernetes после завершения процессов, запущенных в контейнере. Это делается посредством свойства `restartPolicy` в секции `spec` модуля, которое по умолчанию, в других случаях, равняется `Always`. Модули Job не могут использовать эту политику, принятую по умолчанию, поскольку они не предназначены для неограниченного применения. Следовательно, необходимо явно задать политику перезапуска `OnFailure` или `Never`. Такая настройка предотвращает перезапуск контейнера после окончания его работы (не факт, что модуль управляется ресурсом Job).

4.5.3 Просмотр того, как задание управляет модулем

После того как с помощью команды `kubectl create` вы создадите это задание, оно должно быть немедленно запущено:

```
$ kubectl get jobs
NAME       DESIRED  SUCCESSFUL  AGE
batch-job  1        0           2s

$ kubectl get po
NAME             READY  STATUS   RESTARTS  AGE
batch-job-28qf4  1/1    Running  0         4s
```

По истечении двух минут модуль больше не будет появляться в списке модулей, и задание будет помечено как законченное. По умолчанию, если не используется переключатель `--show-all` (или `-a`), законченные модули при выводе списка модулей не показываются:

```
$ kubectl get po -a
NAME             READY  STATUS     RESTARTS  AGE
batch-job-28qf4  0/1    Completed  0         2m
```

Причина, по которой модуль не удаляется после завершения, заключается в том, чтобы позволить вам исследовать его протокол операций, например:

```
$ kubectl logs batch-job-28qf4
Fri Apr 29 09:58:22 UTC 2016 Batch job starting
Fri Apr 29 10:00:22 UTC 2016 Finished succesfully
```

Модуль будет удален, когда вы удалите его или задание, которое его создало. Перед этим давайте еще раз посмотрим на ресурс Job:

```
$ kubectl get job
NAME       DESIRED  SUCCESSFUL  AGE
batch-job  1        1           9m
```

Задание Job отображается как успешно завершенное. Но почему эта информация показана как число, а не как `yes` или `true`? И о чем говорит столбец `DESIRED`?

4.5.4 Запуск нескольких экземпляров модуля в задании

Задания могут быть сконфигурированы для создания нескольких экземпляров модулей и их параллельного или последовательного выполнения. Это делается путем установки свойств `completions` и `parallelism` в секции `spec` ресурса Job.

Последовательное выполнение модулей задания

Если вам нужно, чтобы задание выполнялось более одного раза, то следует установить `completions` в то количество раз, которое нужно, чтобы модуль задания выполнялся. Пример приведен в следующем ниже листинге.

Листинг 4.12. Задание, требующее нескольких завершений: multi-completion-batch-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5        <-- Установка completions равной 5 заставляет это задание выполнить последовательно пять модулей
  template:
    <template is the same as in listing 4.11>
```

Это задание будет выполнять пять модулей один за другим. Сначала оно создает один модуль, а когда контейнер модуля заканчивается, он создает

второй модуль и т. д. до тех пор, пока пять модулей не завершатся успешно. Если один из модулей завершается неуспешно, то задание создает новый модуль, поэтому задание может создать в общей сложности более пяти модулей.

Параллельное выполнение модулей задания

Вместо запуска одного модуля задания один за другим можно также сделать так, чтобы задание выполняло несколько модулей параллельно. С помощью свойства `parallelism` в секции `spec` ресурса Job укажите, сколько модулей разрешено запускать параллельно, как показано в следующем ниже листинге.

Листинг 4.13. Выполнение модулей задания параллельно: multi-completion-parallel-batch-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  parallelism: 2
  template:
    <same as in listing 4.11>
```

Это задание должно обеспечить успешное выполнение пяти модулей

Вплоть до двух модулей могут выполняться параллельно

Установив `parallelism` равным 2, задание создает два модуля и запускает их параллельно:

```
$ kubectl get po
NAME                              READY   STATUS    RESTARTS   AGE
multi-completion-batch-job-lmmnk  1/1     Running   0          21s
multi-completion-batch-job-qx4nq  1/1     Running   0          21s
```

Как только один из них завершится, задание будет выполнять следующий модуль до тех пор, пока пять модулей не закончатся успешно.

Масштабирование задания

Вы можете даже изменить свойство `parallelism` ресурса Job во время выполнения задания. Это похоже на масштабирование ресурса ReplicaSet или ReplicationController и может быть сделано с помощью команды `kubectl scale`:

```
$ kubectl scale job multi-completion-batch-job --replicas 3
job "multi-completion-batch-job" scaled
```

Поскольку вы увеличили свойство `parallelism` с 2 до 3, еще один модуль был сразу же развернут, поэтому теперь работают три модуля.

4.5.5 Ограничение времени, отпускаемого на завершение модуля задания

В отношении заданий нужно обсудить один последний момент. Сколько времени задание должно ждать до завершения работы модуля? Что делать, если модуль застревает и не может закончить вообще (или же он не может закончить достаточно быстро)?

Время модуля можно ограничить, задав свойство `activeDeadlineSeconds` в секции `spec` модуля. Если модуль работает дольше, система попытается его завершить и пометит задание как несработавшее.

> **ПРИМЕЧАНИЕ.** Вы можете сконфигурировать количество повторных попыток выполнения задания, прежде чем оно будет помечено как несработавшее, указав число в поле `spec.backoffLimit` в манифесте ресурса Job. Если его не указать явно, то по умолчанию будет задано 6 раз.

4.6 Планирование выполнения заданий периодически или один раз в будущем

Ресурсы Job запускают свои модули немедленно при создании ресурса Job. Но многие пакетные задания должны выполняться в определенное время в будущем или повторно в заданном интервале. В Linux- и UNIX-подобных операционных системах эти задания более известны как задания cron. Kubernetes их тоже поддерживает.

Cron-задание в Kubernetes конфигурируется путем создания ресурса CronJob. Расписание запуска задания указывается в хорошо известном формате cron, поэтому если вы знакомы с обычными заданиями cron, то вы поймете задания CronJob в Kubernetes за считанные секунды.

В настроенное время Kubernetes создаст ресурс Job в соответствии с шаблоном задания, настроенным в объекте CronJob. При создании ресурса Job будут созданы одна или несколько реплик модуля и запущены в соответствии с шаблоном модуля ресурса Job, как вы узнали в предыдущем разделе. Здесь больше нечего сказать.

Давайте посмотрим, как создавать ресурсы CronJob.

4.6.1 Создание ресурса CronJob

Представьте, что вам нужно запускать пакетное задание из предыдущего примера каждые 15 минут. Для этого создайте ресурс CronJob со следующей спецификацией.

Листинг 4.14. YAML для ресурса CronJob: cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
```

Группа API – batch, версия – v1beta1

```
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
            restartPolicy: OnFailure
            containers:
            - name: main
              image: luksa/batch-job
```

Это задание должно выполняться на 0, 15, 30 и 45 минутах каждого часа каждого дня

Шаблон для ресурсов Job, которые будут созданы этим ресурсом CronJob

Как видите, все не так уж и сложно. Вы указали расписание и шаблон, из которого будут создаваться объекты задания.

Настройка расписания

Если вы незнакомы с форматом расписания cron, вы найдете отличные учебные пособия и пояснения в интернете, но в качестве быстрого введения в курс дела слева направо расписание содержит следующие пять записей:

- минута;
- час;
- день месяца;
- месяц;
- день недели.

В этом примере требуется запускать задание каждые 15 минут, поэтому расписание должно быть «0,15,30,45 * * * *», что означает на отметке в 0, 15, 30 и 45 минут каждого часа (первая звездочка), каждого дня месяца (вторая звездочка), каждого месяца (третья звездочка) и каждого дня недели (четвертая звездочка).

Если бы вместо этого требовалось, чтобы оно запускалось каждые 30 минут, но только в первый день месяца, то вы бы установили расписание «0,30 * 1 * *», и если требуется, чтобы оно работало в 3 часа ночи каждое воскресенье, то вы установите «0 3 * * 0» (последний ноль означает воскресенье).

Настройка шаблона задания

Ресурс CronJob создает ресурсы Job из свойства `jobTemplate`, настраиваемого в секции `spec` ресурса CronJob, поэтому обратитесь к разделу 4.5 для получения дополнительной информации о том, как его настраивать.

4.6.2 Общие сведения о выполнении запланированных заданий

Ресурсы Job будут создаваться из ресурса CronJob приблизительно в запланированное время. Затем Job создает модули.

Может случиться так, что задание или модуль создается и выполняется относительно поздно. У вас может быть жесткое требование, чтобы задание не было запущено слишком отдаленно в течение запланированного времени. В этом случае можно указать крайний срок, задав поле `startingDeadlineSeconds` в секции `spec` ресурса CronJob, как показано в следующем ниже списке.

Листинг 4.15. Определение свойства startingDeadlineSeconds для CronJob

```
apiVersion: batch/v1beta1
kind: CronJob
spec:
  schedule: "0,15,30,45 * * * *"
  startingDeadlineSeconds: 15
  ...
```

Модуль должен начать работу не позднее, чем через 15 секунд после запланированного времени

В примере, приведенном в листинге 4.15, одна из меток времени, когда задание должно запуститься, равняется 10:30:00. Если по какой-либо причине к 10:30:15 оно не запустится, задание не будет запущено и будет показано как Failed.

В обычных условиях CronJob всегда создает всего одно задание для каждого выполнения, настроенного в расписании, но может случиться так, что два задания создаются одновременно или вообще не создаются. Для борьбы с первой проблемой ваши задания должны быть идемпотентными (их выполнение несколько раз, вместо одного раза, не должно привести к нежелательным результатам). Что касается второй проблемы, то убедитесь, что следующий запуск задания выполняет любую работу, которая должна была быть выполнена предыдущим (пропущенным) запуском.

4.7 Резюме

Теперь вы познакомились с тем, как поддерживать работу модулей и переназначать их в случае аварийных завершений работы узлов. Сейчас вы должны знать, что:

- вы можете задать проверку живучести, чтобы Kubernetes перезапускал контейнер, как только он больше не здоров (где приложение определяет, что считать здоровым);
- модули не должны создаваться напрямую, поскольку они не будут создаваться повторно, если они были удалены по ошибке, или если узел, на котором они работают, аварийно завершил работу, или если они были вытеснены из узла;

- контроллеры репликации всегда поддерживают требуемое количество реплик модуля;
- на контроллере репликации горизонтальное масштабирование модулей организуется так же просто, как и изменение количества нужных реплик;
- контроллеры репликации не владеют модулями, и модули при необходимости могут перемещаться между ними;
- контроллер репликации создает новые модули из шаблона модуля. Изменение шаблона не влияет на существующие модули;
- контроллеры репликации должны быть заменены наборами реплик (ReplicaSet) и развертываниями (Deployment), которые обеспечивают такую же функциональность, но с дополнительными мощными возможностями;
- контроллеры репликации и наборы реплик назначают модули случайным узлам кластера, в то время как наборы демонов (DaemonSet) гарантируют, что каждый узел выполняет один экземпляр модуля, определенного в наборе демонов;
- модули, выполняющие пакетную задачу, должны создаваться посредством ресурса Job системы Kubernetes, а не напрямую, или посредством контроллера репликации, или аналогичным объектом;
- задания (Job), которые должны выполняться в будущем, могут быть созданы с помощью ресурсов CronJob.

Глава 5

Службы: обеспечение клиентам возможности обнаруживать модули и обмениваться с ними информацией

Эта глава посвящена:

- созданию ресурсов служб (Service) для обеспечения доступа к группе модулей по единому адресу;
- обнаружению служб в кластере;
- предоставлению внешним клиентам доступа к службам;
- подключению к внешним службам изнутри кластера;
- выполнению проверок на готовность модуля быть частью службы;
- устранению неполадок в службах.

Вы познакомились с модулями и тем, как их разворачивать, посредством наборов реплик и аналогичных ресурсов, которые обеспечивают их бесперебойную работу. Хотя некоторые модули могут делать свою работу независимо от внешнего стимула, многие приложения в наши дни предназначены для того, чтобы откликаться на внешние запросы. Например, в случае микросервисов модули обычно откликаются на HTTP-запросы, поступающие из других модулей внутри кластера или от клиентов за пределами кластера.

Модулям нужно как-то находить другие модули, с тем чтобы потреблять службы, которые те предоставляют. В отличие от мира за пределами Kubernetes, где сисадмин настраивает каждое клиентское приложение в конфигурационных файлах клиента, указывая точный IP-адрес или хостнейм сервера, пре-

доставляющего службу, выполнение того же самого в Kubernetes работать не будет, поскольку:

- *модули эфемерны* – они могут появляться и исчезать в любое время, будь то потому, что модуль был удален из узла, чтобы освободить место для других модулей, либо потому, что кто-то отмасштабировал количество модулей вниз, либо потому, что узел кластера аварийно завершил работу;
- *Kubernetes назначает IP-адрес модулю, после того как модуль был назначен узлу, и до момента его запуска* – следовательно, клиенты не могут знать заранее IP-адрес серверного модуля;
- *горизонтальное масштабирование означает, что несколько модулей могут обеспечивать одну и ту же службу* – каждый из этих модулей имеет свой собственный IP-адрес. Клиенты не должны заботиться о том, сколько модулей поддерживают службу и каковы их IP-адреса. Они не должны вести список всех отдельных IP-адресов модулей. Вместо этого все эти модули должны быть доступны через единый IP-адрес.

Для того чтобы решить эти задачи, Kubernetes предоставляет еще один тип ресурса – службы (Service), которые мы обсудим в этой главе.

5.1 Знакомство со службами

Служба Kubernetes – это ресурс, который вы создаете, чтобы сформировать единую постоянную точку входа в группу модулей, предоставляющих одну и ту же службу. Каждая служба имеет IP-адрес и порт, которые никогда не меняются до тех пор, пока существует служба. Клиенты могут открывать подключения к этому IP-адресу и порту, а затем маршрутизировать эти подключения в один из модулей, поддерживающих эту службу. Благодаря этому клиентам службы не требуется знать расположение отдельных модулей, предоставляющих службу, что позволяет перемещать данные модули по кластеру в любое время.

Объяснение служб на примере

Давайте вернемся к примеру, где у вас имеется фротненд с веб-сервером и бэкенд с базой данных. Может существовать несколько модулей, которые действуют в качестве фротендов, но может быть всего один модуль бэкенда с базой. Для того чтобы эта система функционировала, вам нужно решить 2 задачи:

- внешние клиенты должны подключаться к фронтенду, не заботясь о том, существует ли только один веб-сервер или их сотни;
- модули фронтенда должны подключаться к бэкенду с базой данных. Поскольку база данных работает внутри модуля, с течением времени она может перемещаться по кластеру, что приводит к изменению ее IP-адреса. Вы не хотите перенастраивать фротенд-модули всякий раз, когда база данных перемещается.

Создавая службу для фронтенд-модулей и настраивая ее для доступа извне кластера, вы предоставляете доступ к единому постоянному IP-адресу, через который внешние клиенты могут подключаться к модулям. Аналогичным образом, создавая службу для бэкенд-модуля, вы создаете стабильный адрес для внутреннего модуля. Адрес службы не меняется, даже если меняется IP-адрес модуля. Кроме того, создавая службу, вы также позволяете фронтенд-модулям легко находить внутреннюю службу по ее имени через переменные среды или DNS. Все компоненты вашей системы (две службы, два набора модулей, поддерживающих эти службы, и взаимозависимости между ними) показаны на рис. 5.1.

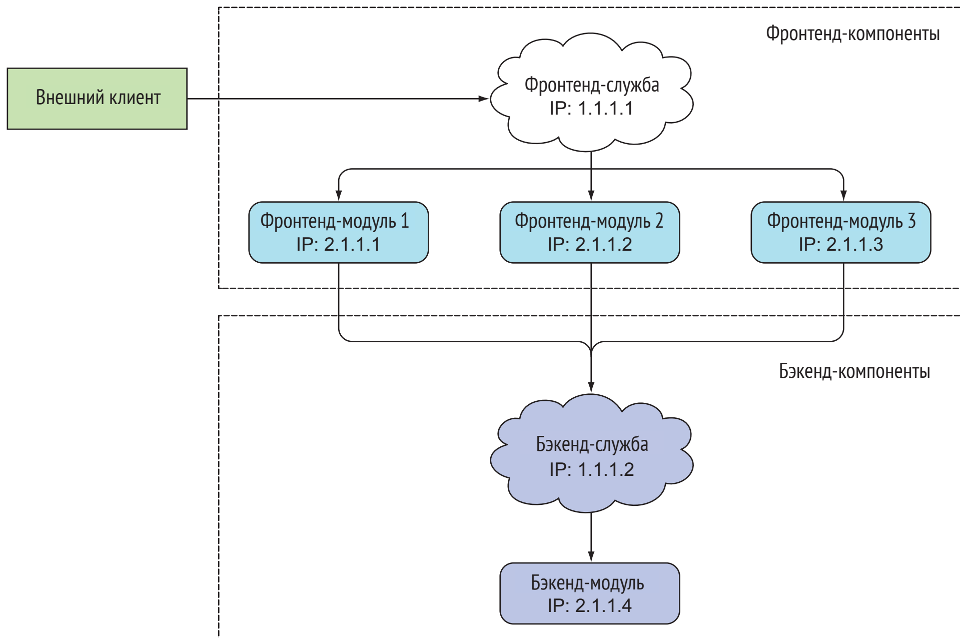


Рис. 5.1. Внутренние и внешние клиенты обычно подключаются к модулям через службы

Теперь вы понимаете основную идею в основе служб. Давайте копнем глубже и сначала увидим, как они могут быть созданы.

5.1.1 Создание служб

Как вы увидели, служба может поддерживаться более чем одним модулем. Подключения к службе сбалансированы по нагрузке на всех поддерживающих модулях. Но как именно определить, какие модули являются частью службы, а какие нет?

Вы, вероятно, помните селекторы меток и то, как они используются в контроллерах репликации и других контроллерах модуля, чтобы определять, какие модули принадлежат одному и тому же набору. Тот же механизм используется и службами, как показано на рис. 5.2.

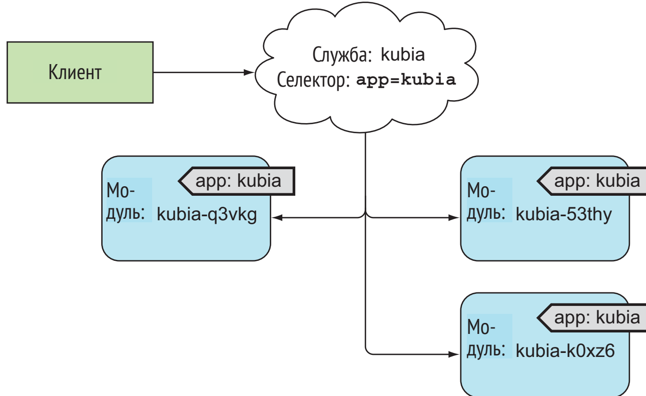


Рис. 5.2. Селекторы меток определяют, какие модули принадлежат службе

В предыдущей главе вы создали контроллер репликации, который затем запустил три экземпляра модуля, содержащего приложение Node.js. Создайте контроллер репликации еще раз и проверьте, что три экземпляра модуля находятся в рабочем состоянии. После этого вы создадите службу для этих трех модулей.

Создание службы с помощью команды kubectl expose

Проще всего создать службу с помощью команды `kubectl expose`, которая уже использовалась в главе 2 для обеспечения доступа к ранее созданному контроллеру репликации. Команда `expose` создала ресурс Service с тем же селектором модуля, что и используемый контроллером репликации, обеспечивая доступ ко всем своим модулям через единый IP-адрес и порт.

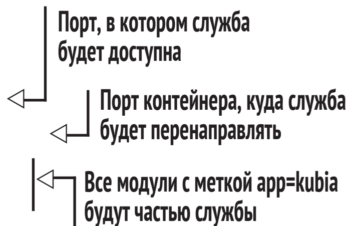
Теперь же, вместо того чтобы использовать команду `expose`, вы создадите службу вручную путем отправки YAML на сервер API Kubernetes.

Создание службы с помощью дескриптора yaml

Создайте файл с именем kubia-svc.yaml, содержимое которого приведено в следующем ниже листинге.

Листинг 5.1. Определение службы: kubia-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```



Вы определяете службу под названием `kubia`, которая будет принимать подключения к порту 80 и маршрутизировать каждое подключение на порт 8080 одного из модулей, который соответствует селектору меток `app=kubia`.

А теперь создайте службу, отправив файл с помощью команды `kubectl create`.

Исследование новой службы

После отправки YAML можно вывести список всех ресурсов службы в вашем пространстве имен и увидеть, что вашей службе был назначен внутрикластерный IP-адрес:

```
$ kubectl get svc
NAME         CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.111.240.1     <none>        443/TCP   30d
kubia        10.111.249.153   <none>        80/TCP    6m      <— Вот ваша служба
```

Этот список показывает, что IP-адресом, назначенным службе, является 10.111.249.153. Поскольку он является кластерным IP-адресом, он доступен только изнутри кластера. Основное назначение служб – обеспечивать доступ к группам модулей другим модулям кластера, но обычно также требуется обеспечивать доступ к службам извне. Вы увидите, как это сделать, позже. На данный момент давайте использовать вашу службу изнутри кластера, и посмотрим, что она делает.

Тестирование службы внутри кластера

Запросы к службе можно отправлять из кластера несколькими способами:

- очевидным способом является создание модуля, который будет отправлять запрос на кластерный IP-адрес службы и регистрировать отклик. Затем можно просмотреть лог модуля, чтобы узнать, каков был отклик службы;
- можно подключиться к одному из узлов Kubernetes посредством команды `ssh` и применить команду `curl`;
- можно выполнить команду `curl` внутри одного из существующих модулей с помощью команды `kubectl exec`.

Давайте перейдем к последнему варианту, чтобы вы также узнали, как запускать команды в существующих модулях.

Удаленное выполнение команд в работающих контейнерах

Команда `kubectl exec` позволяет удаленно запускать произвольные команды внутри существующего контейнера модуля. Это удобно, когда вы хотите изучить содержимое, состояние и/или среду контейнера. Выведите список модулей с помощью команды `kubectl get pods` и выберите один из них в качестве цели для команды `exec` (в следующем ниже примере в качестве цели

я выбрал модуль `kubia-7nog1`). Вам также потребуется получить кластерный IP службы (используя, к примеру, команду `kubectl get svc`). При выполнении следующих ниже команд убедитесь, что имя модуля и IP-адрес службы заменены на ваши собственные:

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
You've hit kubia-gzwli
```

Если ранее вы использовали сетевую команду `ssh` для выполнения команд в удаленной системе, то вы поймете, что команда `kubectl exec` мало чем отличается.

Зачем двойное тире?

Двойное тире (`--`) в команде является сигналом для `kubectl` об окончании командных параметров. Все, что идет после двойного тире, – это команда, которая должна быть выполнена внутри модуля. Использование двойного тире не требуется, если команда не имеет параметров, которые начинаются с тире. Но в вашем случае, если вы не используете двойное тире, параметр `-s` будет интерпретирован как параметр для `kubectl exec` и приведет к следующей ниже странной и крайне обманчивой ошибке:

```
$ kubectl exec kubia-7nog1 curl -s http://10.111.249.153
The connection to the server 10.111.249.153 was refused - did you
     specify the right host or port?
```

Она не имеет ничего общего с отказом вашей службы в подключении. Она связана с тем, что `kubectl` не может подключиться к серверу API по `10.111.249.153` (параметр `-s` используется для указания `kubectl` подключиться к серверу API, отличному от сервера, выбранного по умолчанию).

Давайте рассмотрим, что произошло, когда вы выполнили команду. На рис. 5.3 показана последовательность событий. Вы поручили Kubernetes выполнить команду `curl` внутри контейнера одного из ваших модулей. Команда `curl` отправила HTTP-запрос на IP-адрес службы, который поддерживается тремя модулями. Служебный прокси Kubernetes перехватил подключение, выбрал случайный модуль среди трех модулей и переслал запрос к нему. Node.js, работающий внутри этого модуля, затем обработал запрос и вернул HTTP-отклик, содержащий имя модуля. Затем команда `curl` напечатала отклик в стандартном выводе, который был перехвачен `kubectl` и напечатан в его стандартный поток вывода на локальной машине.

В предыдущем примере вы исполнили команду `curl` как отдельный процесс, но внутри главного контейнера модуля. Это не сильно отличается от обмена фактического главного процесса в контейнере со службой.

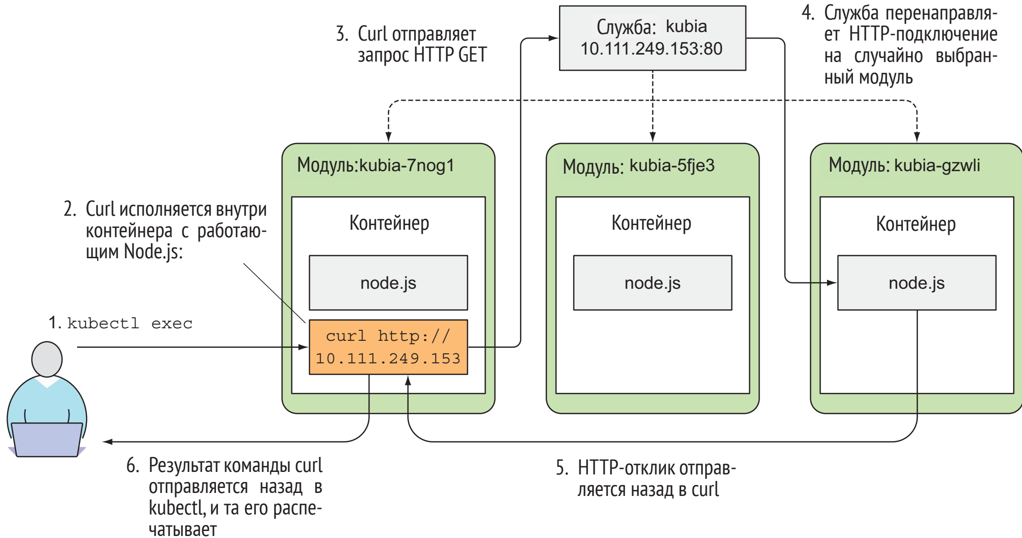


Рис. 5.3. Использование kubectl exec для проверки подключения к службе путем запуска curl в одном из модулей

Конфигурирование сохранения сессий в службах

Если вы исполните ту же команду еще несколько раз, то с каждым вызовом вы будете попадать на другой модуль, потому что служебный прокси обычно передает каждое подключение в случайно выбираемый соответствующий службе модуль, даже если подключения поступают от того же клиента.

Если же вы хотите, чтобы все запросы, сделанные определенным клиентом, всякий раз перенаправлялись в один и тот же модуль, то свойству `sessionAffinity` службы можно присвоить значение `ClientIP` (вместо `None`, которое используется по умолчанию), как показано в следующем ниже листинге.

Листинг 5.2. Пример службы с настроенной сохранностью сессий сеанса ClientIP

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

Это заставляет служебный прокси перенаправлять все запросы, исходящие от того же клиентского IP-адреса на тот же модуль. В качестве упражнения можно создать дополнительную службу с установленным сохранением сессий по принципу `ClientIP` и попытаться отправить ей запросы.

Система Kubernetes поддерживает только два типа сохранения сессий: `None` и `ClientIP`. Вы можете удивиться, что у нее нет параметра сохранения сессий на основе файлов cookie, но следует понимать, что службы Kubernetes не рабо-

тают на уровне HTTP. Службы работают с пакетами TCP и UDP и не заботятся о полезной нагрузке, которую они несут. Поскольку файлы cookie являются конструкцией протокола HTTP, службы о них не знают, что и объясняет, почему сохранение сессий не может быть основано на файлах cookie.

Обеспечение доступа к нескольким портам в одной и той же службе

Ваша служба обеспечивает доступ только к одному порту, однако службы также могут поддерживать несколько портов. Например, если ваши модули прослушивали два порта – скажем, 8080 для HTTP и 8443 для HTTPS, – вы можете использовать одну службу для перенаправления портов 80 и 443 на порты 8080 и 8443 модуля. В таких случаях не требуется создавать две разные службы. Использование единой мультипортовой службы позволяет обеспечивать доступ ко всем портам службы через единый кластерный IP-адрес.

> **ПРИМЕЧАНИЕ.** Во время создания службы с несколькими портами вы должны для каждого порта указать имя.

Свойство `spec` ресурса многопортовой службы показано в следующем ниже листинге.

Листинг 5.3. Указание нескольких портов в определении службы

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8080
  - name: https
    port: 443
    targetPort: 8443
  selector:
    app: kubia
```

Порт 80 увязан с портом 8080 модуля

Порт 443 увязан с портом 8443 модуля

Селектор меток всегда применяется ко всей службе в целом

> **ПРИМЕЧАНИЕ.** Селектор меток применяется ко всей службе в целом – он не может быть настроен для каждого порта индивидуально. Если вы хотите, чтобы разные порты увязывались с разными подмножествами модулей, необходимо создать две службы.

Поскольку ваши модули `kubia` не прослушивают на нескольких портах, создание мультипортовой службы и мультипортового модуля оставлено для вас в качестве упражнения.

Использование именованных портов

Во всех этих примерах вы ссылаетесь на целевой порт по его номеру, но вы также можете дать порту каждого модуля имя и ссылаться на него по имени в спецификации службы. Это делает спецификацию службы немного яснее, в особенности если номера портов не стандартны.

Например, предположим, что в модуле определены имена портов, как показано в следующем ниже листинге.

Листинг 5.4. Задание имен портов в определении модуля

```
kind: Pod
spec:
  containers:
  - name: kubia
    ports:
    - name: http              # Порт 8080 контейнера назван http
      containerPort: 8080
    - name: https             # Порт 8443 назван https
      containerPort: 8443
```

Затем в спецификации службы можно обращаться к этим портам по имени, как показано в следующем ниже листинге.

Листинг 5.5. Ссылка на именованные порты в службе

```
apiVersion: v1
kind: Service
spec:
  ports:
  - name: http               # Порт 80 увязан с портом контейнера под название http
    port: 80
    targetPort: http
  - name: https              # Порт 443 увязан с портом контейнера, имя которого https
    port: 443
    targetPort: https
```

Но зачем вообще нужно беспокоиться об именовании портов? Самое большое преимущество именования портов заключается в том, что оно позже позволяет вам изменять номера портов без необходимости изменять спецификацию службы. Ваш модуль в настоящее время использует порт 8080 для `http`, но что, если вы позже решите переместить его в порт 80?

Если вы используете именованные порты, то вам нужно лишь изменить номер порта в спецификации модуля (сохраняя при этом имя порта без изменений). Когда вы запускаете модули с новыми портами, клиентские подключения будут перенаправлены на соответствующие номера портов в зависимости от модуля, получающего подключение (порт 8080 на старых модулях и порт 80 на новых).

5.1.2 Обнаружение служб

Создав службу, теперь у вас есть один стабильный IP-адрес и порт, который вы можете использовать для доступа к своим модулям. Этот адрес будет оставаться неизменным в течение всего срока службы. Модули, стоящие за этой службой, могут приходить и уходить, их IP-адреса могут меняться, их количество может увеличиваться или уменьшаться, но они всегда будут доступны через единый и постоянный IP-адрес службы.

Но каким образом клиентские модули знают IP и порт службы? Нужно ли сначала создать службу, а затем вручную найти ее IP-адрес и передать IP-адрес параметрам конфигурации клиентского модуля? Не совсем. Kubernetes предоставляет клиентским модулям возможность обнаруживать IP-адрес и порт службы.

Обнаружение служб с помощью переменных среды

При запуске модуля Kubernetes инициализирует набор переменных среды, указывающих на каждую службу, существующую в данный момент. Если вы создаете службу перед созданием клиентских модулей, процессы в этих модулях могут получить IP-адрес и порт службы путем проверки их переменных среды.

Давайте посмотрим, как выглядят эти переменные среды, исследовав среду одного из ваших работающих модулей. Вы уже выяснили, что команда `kubectl exec` применяется для выполнения команды в модуле, но поскольку служба была создана только после создания модулей, переменные среды для службы еще могли быть не заданы. Сначала вам нужно решить этот вопрос.

Прежде чем можно будет просмотреть переменные среды службы, сначала необходимо удалить все модули и разрешить контроллеру репликации создать новые. Возможно, вы помните, что можете удалять все модули, не указывая их имена, как показано ниже:

```
$ kubectl delete po --all
pod "kubia-7nog1" deleted
pod "kubia-bf50t" deleted
pod "kubia-gzwli" deleted
```

Теперь вы можете вывести список новых модулей (уверен, что вы знаете, как это сделать) и выбрать один модуль для команды `kubectl exec` в качестве цели. После выбора целевого модуля можно вывести список переменных среды, выполнив команду `env` внутри контейнера, как показано в следующем ниже листинге.

Листинг 5.6. Переменные среды службы в контейнере

```
$ kubectl exec kubia-3inly env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kubia-3inly
```

```
KUBERNETES_SERVICE_HOST=10.111.240.1
KUBERNETES_SERVICE_PORT=443
...
KUBIA_SERVICE_HOST=10.111.249.153      <-- Вот кластерный IP службы
KUBIA_SERVICE_PORT=80                  <-- А вот порт, по которому служба доступна
...
```

В кластере определены две службы: `kubernetes` и `kubia` (вы видели это ранее с помощью команды `kubectl get svc`); следовательно, в списке есть два набора переменных среды, связанных со службами. Среди переменных, относящихся к службе `kubia`, созданной в начале главы, вы увидите переменные среды `KUBIA_SERVICE_HOST` и `KUBIA_SERVICE_PORT`, которые содержат соответственно IP-адрес и порт службы `kubia`.

Возвращаясь к примеру с фронтенд- и бэкенд-модулями, с которого мы начали эту главу, когда у вас есть фронтенд-модуль, который требует использования бэкенд-модуля с сервером базы данных, вы можете обеспечить доступ к бэкенд-модулю через службу под названием `backend-database`, а потом уже дать фронтенд-модулю просматривать ее IP-адрес и порт через переменные среды `BACKEND_DATABASE_SERVICE_HOST` и `BACKEND_DATABASE_SERVICE_PORT`.

> **ПРИМЕЧАНИЕ.** Когда имя службы используется в качестве префикса в имени переменной окружения, символы тире в имени службы преобразуются в символы подчеркивания, и все буквы преобразуются в верхний регистр.

Переменные среды – это один из способов найти IP и порт службы, но разве это не является обычной работой DNS? Почему система Kubernetes не включает DNS-сервер и не позволяет искать IP-адреса служб через DNS? Как оказалось, именно это она и делает!

Обнаружение служб через DNS

Помните, в главе 3 вы выводили список модулей в пространстве имен `kube-system`? Один из модулей назывался `kube-dns`. Пространство имен `kube-system` включает соответствующую службу с тем же именем.

Как следует из названия, модуль запускает DNS-сервер, для использования которого автоматически настраиваются все остальные модули, работающие в кластере (Kubernetes делает это, изменяя файл `/etc/resolv.conf` каждого контейнера). Любой DNS-запрос, выполняемый процессом, запущенным в модуле, будет обрабатываться собственным DNS-сервером Kubernetes, который знает все службы, работающие в вашей системе.

> **ПРИМЕЧАНИЕ.** С помощью свойства `dnsPolicy` в поле `spec` ресурса каждого модуля можно настроить, будет модуль использовать внутренний DNS-сервер или нет.

Каждая служба получает DNS-запись во внутреннем DNS-сервере, и клиентские модули, которые знают имя службы, могут обращаться к ней через полностью квалифицированное доменное имя (FQDN) вместо использования переменных среды.

Подключение к службе через FQDN

Возвращаясь к примеру с фронтенд-модулями, фронтенд-модуль может подключиться к службе бэкенд-базы данных, открыв подключение со следующим ниже полным доменным именем:

```
backend-database.default.svc.cluster.local
```

где `backend-database` соответствует имени службы, `default` обозначает пространство имен, в котором определена служба, и `svc.cluster.local` – это настраиваемый доменный суффикс кластера, используемый во всех именах локальных служб кластера.

> **ПРИМЕЧАНИЕ.** Клиент должен знать номер порта службы. Если служба использует стандартный порт (например, 80 для HTTP или 5432 для Postgres), это не должно быть проблемой. В противном случае клиент может получить номер порта из переменной среды.

Процедура подключения к службе может оказаться даже проще. Когда модуль интерфейса находится в том же пространстве имен, что и модуль базы данных, вы можете опустить суффикс `svc.cluster.local` и даже пространство имен. В результате вы можете ссылаться на службу просто как `backend-database`. Невероятно просто, не правда?

Давайте попробуем. Вы попытаетесь получить доступ к службе `kubia` через ее полное доменное имя, а не IP-адрес. Опять-таки, вам нужно сделать это внутри существующего модуля. Вы уже знаете, как применять команду `kubectl exec` для выполнения одной команды в контейнере модуля, но на этот раз, вместо того чтобы выполнять команду `curl` напрямую, вы запустите оболочку `bash`, чтобы затем запустить несколько команд в контейнере. Это похоже на то, что вы делали в главе 2 при входе в контейнер, который вы запустили с помощью Docker, используя команду `docker exec -it bash`.

Запуск оболочки в контейнере модуля

Для запуска `bash` (или любой другой оболочки) внутри контейнера модуля можно использовать команду `kubectl exec`. Благодаря этому вы можете исследовать контейнер столько, сколько захотите, без необходимости выполнять `kubectl exec` для каждой команды, которую вы хотите применить.

> **ПРИМЕЧАНИЕ.** Для этой работы двоичный исполняемый файл оболочки должен быть доступен в образе контейнера.

Для того чтобы правильно использовать оболочку, необходимо в команду `kubectl exec` передать параметр `-it`:

```
$ kubectl exec -it kubia-3inly bash
root@kubia-3inly:/#
```

Теперь вы находитесь внутри контейнера. Команду `curl` можно использовать для доступа к службе `kubia` любым из следующих ниже способов:

```
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local
You've hit kubia-5asi2

root@kubia-3inly:/# curl http://kubia.default
You've hit kubia-3inly

root@kubia-3inly:/# curl http://kubia
You've hit kubia-8awf3
```

Попасть в свою службу можно, используя в запрашиваемом URL имя службы в качестве хостнейма. Вследствие того, как настроен распознаватель DNS внутри контейнера каждого модуля, можно опустить пространство имен и суффикс `svc.cluster.local`. Обратитесь к файлу `/etc/resolv.conf` в контейнере, и вы поймете:

```
root@kubia-3inly:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local ...
```

Почему нельзя пропинговать IP-адрес службы

Еще кое-что, прежде чем мы двинемся дальше. Сейчас вы знаете, как создавать службы, поэтому скоро создадите свои. Но что, если по какой-то причине вы не можете получить доступ к своим службам?

Вероятно, вы попытаетесь выяснить, в чем проблема, войдя в существующий модуль и попытавшись получить доступ к службе, как в последнем примере. Затем, если вы по-прежнему не можете получить доступ к службе с помощью простой команды `curl`, возможно, вы попытаетесь пропинговать IP-адрес службы, чтобы убедиться, что она работает. Давайте попробуем:

```
root@kubia-3inly:/# ping kubia
PING kubia.default.svc.cluster.local (10.111.249.153): 56 data bytes
^C--- kubia.default.svc.cluster.local ping statistics ---
54 packets transmitted, 0 packets received, 100% packet loss
```

Хм. Диагностика службы командой `curl` работает, но пропинговать ее не получается. Причина в том, что кластерный IP службы является виртуальным IP-адресом и имеет смысл только в сочетании с портом службы. Мы объясним, что это означает и как работают службы, в главе 11. Я хотел упомянуть это здесь, потому что это первое, что пользователи делают, когда они пытаются

отладить нарушенную службу, и данная ситуация застает большинство из них врасплох.

5.2 Подключение к службам, находящимся за пределами кластера

До сих пор мы говорили о службах, поддерживаемых одним или несколькими модулями, работающими внутри кластера. Но существуют случаи, когда через функционал служб Kubernetes требуется обеспечить доступ к внешним службам. Вместо того чтобы служба перенаправляла подключения к модулям в кластере, требуется, чтобы она перенаправляла на внешние IP-адреса и порты.

Это позволяет использовать преимущества и балансировки нагрузки служб и обнаружения служб. Клиентские модули, работающие в кластере, могут подключаться к внешней службе так же, как и к внутренним службам.

5.2.1 Знакомство с конечными точками служб

Прежде чем перейти к тому, как это сделать, позвольте сначала остановиться на службах подробнее. Службы не связываются с модулями напрямую. Вместо этого посередине находится ресурс конечных точек (Endpoints). Возможно, вы уже обратили внимание на конечные точки, если использовали команду `kubectl describe` в службе, как показано в следующем ниже листинге.

Листинг 5.7. Подробные сведения о службе, выводимой с помощью команды kubectl describe

```
$ kubectl describe svc kubia
Name:               kubia
Namespace:          default
Labels:             <none>
Selector:           app=kubia        <- Селектор модулей службы используется для создания списка конечных точек
Type:               ClusterIP
IP:                 10.111.249.153
Port:               <unset> 80/TCP
Endpoints:          10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080   <- Список IP-адресов и портов модуля, представляющих конечные точки этой службы
Session Affinity:   None
No events.
```

Ресурс Endpoints (да, конечные точки во множественном числе) – это список IP-адресов и портов, предоставляющих доступ к службе. Ресурс конечных точек похож на любой другой ресурс Kubernetes, поэтому вы можете вывести его основную информацию с помощью команды `kubectl get`:

```
$ kubectl get endpoints kubia
NAME   ENDPOINTS                                         AGE
kubia  10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080  1h
```

Несмотря на то что селектор модулей определен в поле спецификации службы, он не используется непосредственно при перенаправлении входящих подключений. Вместо этого селектор используется для построения списка IP-адресов и портов, который затем хранится в ресурсе конечных точек. Когда клиент подключается к службе, служебный прокси выбирает одну из этих пар IP-адресов и портов и перенаправляет входящее подключение на сервер, прослушивающий в этом месте.

5.2.2 Настройка конечных точек службы вручную

Возможно, вы уже это поняли, но все же – если конечные точки службы отделены от службы, то их можно конфигурировать и обновлять вручную.

Если вы создаете службу без селектора модулей, то соответственно Kubernetes не создаст ресурс конечных точек (в конце концов, без селектора он не может знать, какие модули включать в службу). На вас лежит обязанность создать ресурс Endpoints и указать список конечных точек для службы.

Для создания службы с управляемыми вручную конечными точками нужно создать ресурсы Service и Endpoints.

Создание службы без селектора

Сначала вы создадите YAML для самой службы, как показано в следующем ниже листинге.

Листинг 5.8. Служба без селектора модулей: external-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  ports:
  - port: 80
```

Имя службы должно совпадать с именем объекта Endpoints (см. следующий ниже листинг)

В этой службе селектор не определен

Вы определяете службу под названием `external-service`, которая будет принимать входящие подключения на порту 80. Вы не определили для службы селектор модулей.

Создание ресурса конечных точек для службы без селектора

Конечные точки (Endpoints) представляют собой отдельный ресурс, а не атрибут службы. Поскольку вы создали службу без селектора, соответствующий ресурс Endpoints не был создан автоматически, поэтому его создание зависит от вас. В следующем ниже листинге показан манифест YAML.

Листинг 5.9. Созданный вручную ресурс Endpoints: external-service-endpoints.yaml

```
apiVersion: v1
kind: Endpoints
```

```
metadata:
  name: external-service       <- Имя объекта Endpoints должно совпадать
                                  с именем службы (см. предыдущий листинг)
subsets:
  - addresses:
    - ip: 11.11.11.11          <- IP-адреса конечных точек, на которые
    - ip: 22.22.22.22             служба будет перенаправлять подключения
    ports:
    - port: 80                 <- Целевой порт конечных точек
```

Объект Endpoints должен иметь то же имя, что и служба, и содержать для службы список целевых IP-адресов и портов. После того как служба и ресурс Endpoints будут отправлены на сервер, служба будет готова к использованию, как любая обычная служба с селектором модулей. Контейнеры, созданные после создания службы, будут включать переменные среды для службы, и все подключения с ее парой IP:port будут балансироваться между конечными точками службы.

На рис. 5.4 показаны три модуля, подключающихся к службе с внешними конечными точками.

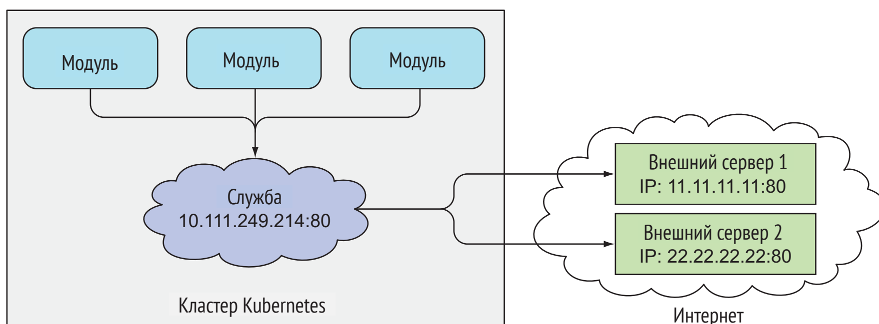


Рис. 5.4. Модули, потребляющие службу с двумя внешними конечными точками

Если позднее вы решите перенести внешнюю службу в модули, работающие внутри Kubernetes, вы можете добавить селектор в службу, тем самым сделав ее конечные точки управляемыми автоматически. То же самое верно и в обратном направлении – путем удаления селектора из службы Kubernetes прекращает обновление конечных точек. Это означает, что IP-адрес службы может оставаться постоянным при изменении фактической реализации службы.

5.2.3 Создание псевдонима для внешней службы

Вместо предоставления доступа к внешней службе путем ручной настройки конечных точек службы более простой метод позволяет ссылаться на внешнюю службу по ее полностью квалифицированному доменному имени (FQDN).

Создание службы ExternalName

Для того чтобы создать службу, которая служит псевдонимом для внешней службы, необходимо создать ресурс Service с полем `type`, установленным в

`ExternalName`. Например, предположим, что общедоступный API имеется по адресу `api.somecompany.com`. Как показано в следующем ниже листинге, вы можете определить службу, которая на него указывает.

Листинг 5.10. Служба с типом ExternalName: external-service-externalname.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName                         <- Типу службы присвоено значение ExternalName
  externalName: someapi.somecompany.com      <- Полное доменное имя фактической службы
  ports:
  - port: 80
```

После создания службы вместо использования фактического полного доменного имени службы модули могут подключаться к внешней службе через доменное имя `external-service.default.svc.cluster.local` (или даже `external-service`). В результате фактическое имя службы и ее местоположение будут скрыты от модулей, которые потребляют службу, позволяя вам модифицировать определение службы и указывать ее на другую службу в любое время позже путем изменения только атрибута `externalName` или путем возврата типа обратно в `ClusterIP` и создания объекта Endpoints для службы – вручную либо путем указания селектора меток на службе и давая ей создаваться автоматически.

Службы `ExternalName` реализуются исключительно на уровне DNS – для службы создается простая DNS-запись `CNAME`. Поэтому клиенты, подключающиеся к службе, будут подключаться к внешней службе напрямую, полностью минуя служебный прокси. По этой причине эти виды служб не получают кластерный IP-адрес.

> **ПРИМЕЧАНИЕ.** Ресурсная запись CNAME указывает на полное доменное имя вместо числового IP-адреса.

5.3 Предоставление внешним клиентам доступа к службам

До сих пор мы говорили только о том, как службы могут потребляться модулями изнутри кластера. Но вам также потребуется обеспечивать доступ к некоторым службам, таким как фронтенд-веб-серверы, извне, чтобы внешние клиенты могли получать к ним доступ, как показано на рис. 5.5.

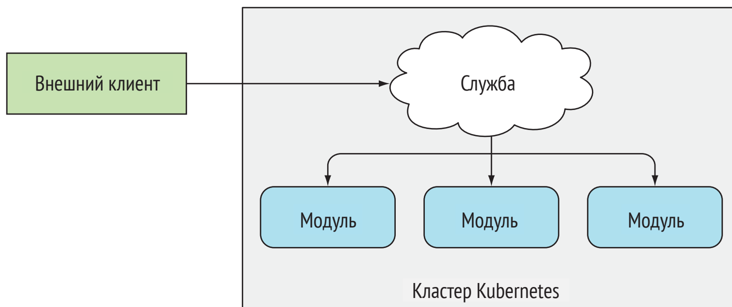


Рис. 5.5. Предоставление внешним клиентам доступа к службе

Есть несколько способов, как сделать службу доступной извне:

- *присвоить типу службы значение* `NodePort` – для службы `NodePort` каждый узел кластера открывает порт на самом узле (отсюда и название) и перенаправляет трафик, получаемый на этом порту, в базовую службу. Служба доступна не только через внутренний IP-адрес и порт кластера, но и через выделенный порт на всех узлах;
- *присвоить типу службы значение* `LoadBalancer`, *расширение типа* `NodePort` – это делает службу доступной через выделенный балансировщик нагрузки, зарезервированный из облачной инфраструктуры, на которой работает Kubernetes. Балансировщик нагрузки перенаправляет трафик на порт узла во всех узлах. Клиенты подключаются к службе через IP-адрес балансировщика нагрузки;
- *создать ресурс Ingress, радикально отличающийся механизм предоставления доступа к нескольким службам через единый IP-адрес* – он работает на уровне HTTP (сетевой уровень 7) и, следовательно, может предложить больше возможностей, чем службы уровня 4. Мы объясним ресурсы Ingress в разделе 5.4.

5.3.1 Использование службы NodePort

Первый способ предоставления внешним клиентам доступа к набору модулей – это создание службы и установка для нее типа `NodePort`. Создавая службу `NodePort`, вы заставляете Kubernetes резервировать порт на всех его узлах (во всех из них используется один и тот же номер порта) и перенаправлять входящие подключения в модули, которые являются частью службы.

Это похоже на обычную службу (их фактический тип – `ClusterIP`), но к службе `NodePort` можно получить доступ не только через внутренний кластерный IP-адрес службы, но и через IP-адрес любого узла и зарезервированный порт узла.

Это будет понятнее, когда вы попытаетесь взаимодействовать со службой `NodePort`.

Создание службы NodePort

Теперь вы создадите службу `NodePort`, чтобы посмотреть, как ее можно использовать. В следующем ниже листинге показан YAML для этой службы.

Листинг 5.11. Определение службы NodePort: kubia-svc-nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30123
  selector:
    app: kubia
```

Задать тип службы NodePort (type: NodePort)

Это порт внутреннего кластерного IP службы (port: 80)

Это целевой порт модулей, связанных с этой службой (targetPort: 8080)

Служба будет доступна через порт 30123 каждого вашего узла кластера (nodePort: 30123)

Задайте тип `NodePort` и укажите порт узла, к которому должна быть привязана эта служба на всех узлах кластера. Указание порта не является обязательным. Если вы его опустите, Kubernetes выберет случайный порт.

> **ПРИМЕЧАНИЕ.** При создании службы в GKE `kubectl` распечатает предупреждение о том, что необходимо настроить правила брандмауэра. Вскоре мы увидим, как это делается.

Исследование вашей службы NodePort

Давайте посмотрим основную информацию о вашей службе, чтобы узнать о ней побольше:

```
$ kubectl get svc kubia-nodeport
NAME             CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubia-nodeport   10.111.254.223   <nodes>       80:30123/TCP   2m
```

Взгляните на столбец `EXTERNAL-IP`. Он показывает `<nodes>`, говоря, что служба доступна через IP-адрес любого узла кластера. Столбец `PORT(S)` показывает и внутренний порт кластерного IP (`80`), и порт узла (`30123`). Служба доступна по следующим адресам:

- `10.11.254.223:80`;
- `<IP 1-го узла>: 30123`;
- `<IP 2-го узла>: 30123` и т. д.

На рис. 5.6 показана служба, доступ к которой предоставляется на порту 30123 обоих ваших узлов кластера (это применимо, в случае если вы работаете на GKE; Minikube имеет всего один узел, но принцип тот же). Входящее

подключение с одним из этих портов будет перенаправлено на случайно выбранный модуль, могущий быть или не быть модулем, работающим на узле, к которому выполняется подключение.

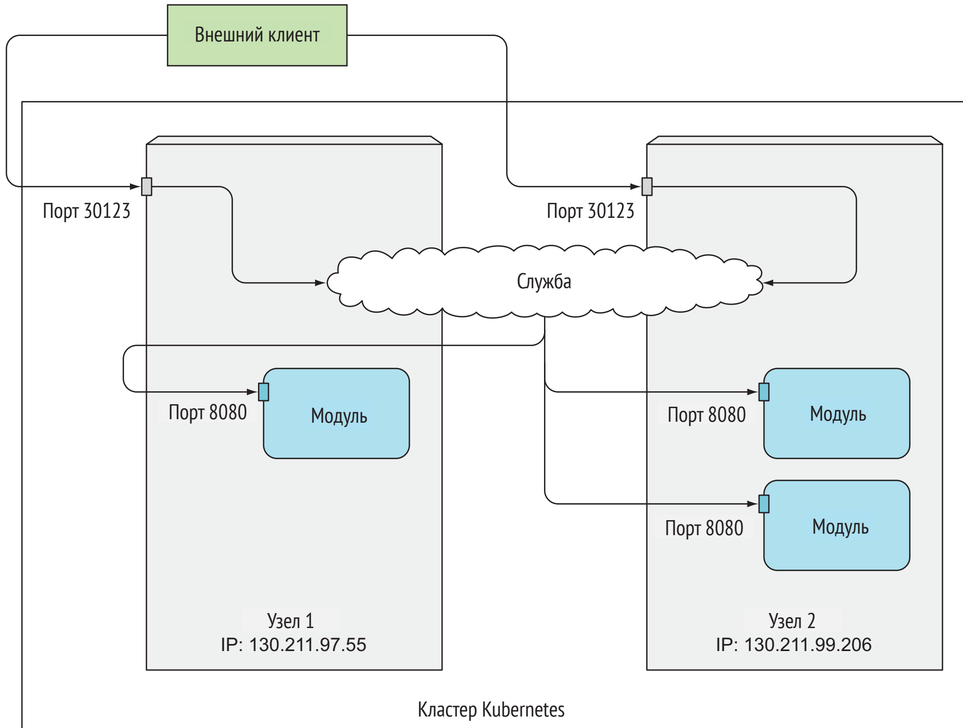


Рис. 5.6. Внешний клиент, соединяющийся со службой NodePort через узел 1 либо через узел 2

Подключение, полученное на порту 30123 первого узла, могло быть перенаправлено либо к модулю, работающему на первом узле, либо к одному из модулей, работающих на втором узле.

Изменение правил брандмауэра для обеспечения внешним клиентам доступа к службе NodePort

Как я уже упоминал ранее, прежде чем обратиться к службе через порт узла, необходимо настроить брандмауэры облачной платформы Google (Google Cloud Platform), чтобы разрешить внешние подключения к узлам на этом порту. Сейчас вы это сделаете:

```
$ gcloud compute firewall-rules create kubia-svc-rule --allow=tcp:30123
Created [https://www.googleapis.com/compute/v1/projects/kubia-
    1295/global/firewalls/kubia-svc-rule].
NAME            NETWORK   SRC_RANGES   RULES        SRC_TAGS   TARGET_TAGS
kubia-svc-rule  default   0.0.0.0/0    tcp:30123
```

Вы можете обратиться к службе через порт 30123 одного из IP-адресов узла. Но сначала нужно выяснить IP-адрес узла. Смотрите вставку ниже о том, как это сделать.

Использование JSONPath для получения IP-адресов всех своих узлов

Вы можете найти IP в дескрипторах JSON или YAML узлов. Но вместо перелопачивания относительно большого документа JSON можно поручить инструменту `kubectl` распечатать лишь IP узла, а не все определение службы:

```
$ kubectl get nodes -o jsonpath='{.items[*].status.
➥ addresses[?(@.type=="ExternalIP")].address}'
130.211.97.55 130.211.99.206
```

Указав параметр `jsonpath`, вы поручаете инструменту `kubectl` выводить только ту информацию, которую вы хотите. Вы, вероятно, знакомы с XPath и как он используется с XML. JSONPath в основном представляет собой XPath для JSON. JSONPath в предыдущем примере поручает `kubectl` сделать следующее:

- просмотреть все элементы в атрибуте `items`;
- по каждому элементу войти в атрибут `status`;
- отфильтровать элементы атрибута `addresses`, принимая только те, у которых атрибут `type` имеет значение `ExternalIP`;
- наконец, вывести атрибут `address` отфильтрованных элементов.

Для того чтобы узнать больше о том, как JSONPath используется с `kubectl`, обратитесь к документации на http://kubernetes.io/docs/user-guide/jsonpath.

После того как вы знаете IP-адреса своих узлов, вы можете попробовать обратиться через них к своей службе:

```
$ curl http://130.211.97.55:30123
You've hit kubia-ym8or
$ curl http://130.211.99.206:30123
You've hit kubia-xueq1
```

> **СОВЕТ.** При использовании Minikube можно легко получить доступ к службам NodePort через браузер, выполнив `minikube service <имя-службы> [-n <пространство-имен>]`.

Как вы можете видеть, ваши модули теперь доступны для всего интернета через порт 30123 на любом из ваших узлов. Не имеет значения, на какой узел клиент отправляет запрос. Но если вы направите своих клиентов только на первый узел, то при аврийном сбое этого узла ваши клиенты больше не смогут получать доступ к службе. Вот почему имеет смысл поставить перед

узлами балансировщик нагрузки, который будет обеспечивать распределение запросов по всем здоровым узлам и никогда не будет отправлять их на узел, который в данный момент находится в отключенном состоянии.

Если кластер Kubernetes его поддерживает (что в основном верно при развертывании Kubernetes в облачной инфраструктуре), то балансировщик нагрузки может быть зарезервирован автоматически, создав службу `LoadBalancer` вместо службы `NodePort`. Мы это рассмотрим далее.

5.3.2 Обеспечение доступа к службе через внешнюю подсистему балансировки нагрузки

Кластеры Kubernetes, работающие на облачных провайдерах, обычно поддерживают автоматическое резервирование балансировщика нагрузки из облачной инфраструктуры. От вас требуется только установить тип службы `LoadBalancer` вместо `NodePort`. Балансировщик нагрузки будет иметь собственный уникальный общедоступный IP-адрес и перенаправлять все подключения к службе. Следовательно, вы сможете получить доступ к службе через IP-адрес балансировщика нагрузки.

Если Kubernetes работает в среде, которая не поддерживает службы `LoadBalancer`, подсистема балансировки нагрузки не будет создана, но ваша служба будет по-прежнему работать как служба `NodePort`. Это вызвано тем, что служба `LoadBalancer` является расширением службы `NodePort`. Вам следует запустить этот пример на движке Google Kubernetes Engine, который поддерживает службы `LoadBalancer`. Minikube же их не поддерживает, по крайней мере не тогда, когда писался этот текст.

Создание службы LoadBalancer

Для того чтобы создать службу с балансировщиком нагрузки, создайте службу из манифеста YAML, как показано в приведенном ниже листинге.

Листинг 5.12. Служба с типом LoadBalancer: kubia-svc-loadbalancer.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-loadbalancer
spec:
  type: LoadBalancer      <- Этот вид службы получает балансировщик
  ports:                     нагрузки из инфраструктуры, в которой
  - port: 80                 размещен кластер Kubernetes
    targetPort: 8080
  selector:
    app: kubia
```

Тип службы задан как `LoadBalancer` вместо `NodePort`. Вы не указываете конкретный порт узла, хотя могли бы (вместо этого вы позволяете системе Kubernetes выбрать самой).

Подключение к службе через балансировщик нагрузки

После создания службы потребуется некоторое время на создание подсистемы балансировки нагрузки и записи ее IP-адреса в объект Service облачной. После этого IP-адрес будет указан как внешний IP-адрес службы:

```
$ kubectl get svc kubia-loadbalancer
NAME                 CLUSTER-IP       EXTERNAL-IP      PORT(S)        AGE
kubia-loadbalancer   10.111.241.153   130.211.53.173   80:32143/TCP   1m
```

В этом случае балансировщик нагрузки доступен по IP-адресу 130.211.53.173, поэтому теперь можно обратиться к службе по этому IP-адресу:

```
$ curl http://130.211.53.173
You've hit kubia-xueq1
```

Великолепно! Как вы могли заметить, на этот раз вам не нужно возиться с брандмауэрами, как вы делали бы раньше со службой `NodePort`.

> **Сохранение сессий и веб-браузеры**
>
> Поскольку ваша служба теперь доступна извне, вы можете попробовать получить к ней доступ с помощью веб-браузера. Вы увидите что-то, что может показаться вам странным, – браузер будет каждый раз попадать в один и тот же модуль. Изменилось ли свойство сохранения сессий в службе за это время? С помощью команды `kubectl explain` вы можете перепроверить, что сохранение сессий службы по-прежнему установлено в None, тогда почему разные запросы браузера не попадают в разные модули, как это происходит при использовании `curl`?
>
> Позвольте мне объяснить, что происходит. Браузер использует постоянное (keep-alive) подключение и отправляет все свои запросы через одно подключение, в то время как `curl` каждый раз открывает новое подключение. Службы работают на уровне подключения, поэтому при первом открытии подключения к службе выбирается случайный модуль, а затем все сетевые пакеты, принадлежащие этому подключению, отправляются в данный модуль. Даже если сохранение сессий установлено в None, пользователи всегда будут попадать в один и тот же модуль (пока подключение не будет закрыто).

Взгляните на рис. 5.7, чтобы видеть, как HTTP-запросы доставляются к модулю. Внешние клиенты (в вашем случае `curl`) подключаются к порту 80 подсистемы балансировки нагрузки и перенаправляются на неявно назначенный порт узла на одном из узлов. Оттуда подключение перенаправляется на один из экземпляров модуля.

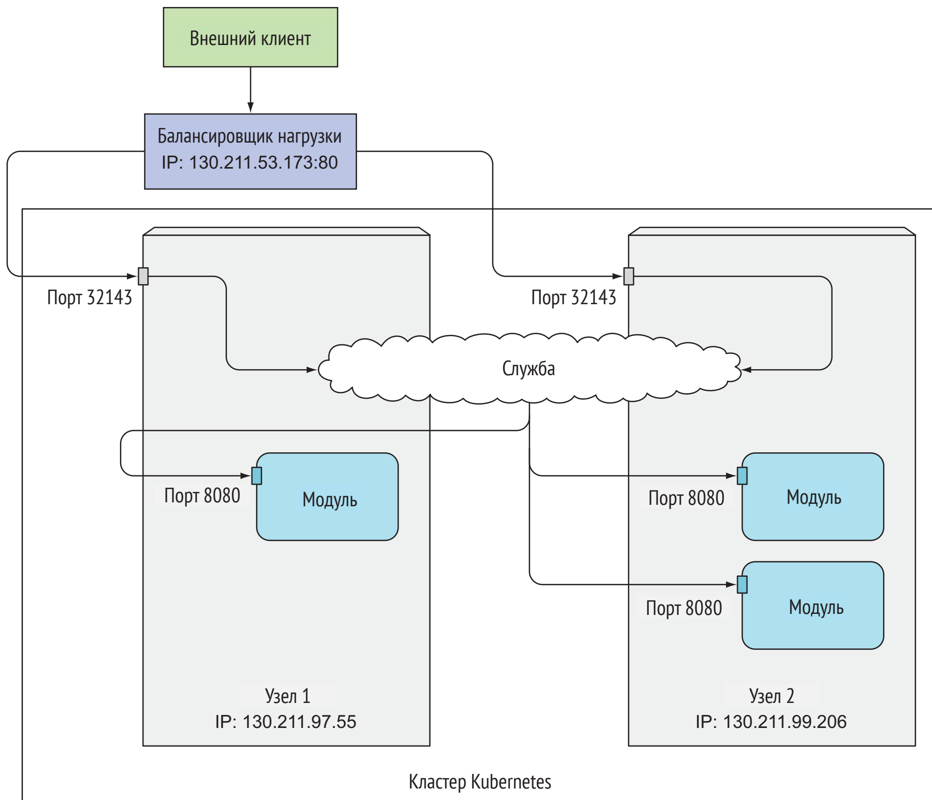


Рис. 5.7. Внешний клиент подключается к службе балансировки нагрузки

Как уже отмечалось, служба с типом `LoadBalancer` – это служба `NodePort` с дополнительной подсистемой балансировки нагрузки, предоставляемой инфраструктурой. Если для вывода дополнительной информации о службе применить команду `kubectl describe`, то вы увидите, что для службы выбран порт узла. Если бы вы открыли брендмауэр для этого порта так, как вы сделали в предыдущем разделе о службах `NodePort`, то вы бы также смогли обратиться к службе через IP-адрес узла.

> **СОВЕТ.** Если вы используете Minikube, даже если подсистема балансировки нагрузки никогда не будет зарезервирована, вы все равно сможете получить доступ к службе через порт узла (по IP-адресу виртуальной машины Minikube).

5.3.3 Особенности внешних подключений

Следует учесть несколько аспектов, связанных с внешними подключениями к службам.

Понимание и предотвращение ненужных сетевых переходов

Когда внешний клиент подключается к службе через порт узла (это также относится к случаям, когда он сначала проходит через балансировщик нагруз-

ки), случайно выбранный модуль может не работать на том же узле, который получил подключение. Для достижения модуля потребуется дополнительный сетевой переход (hop), но это не всегда желательно.

Этот дополнительный переход можно предотвратить, настроив службу для перенаправления внешнего трафика только на те модули, которые работают на узле, получившем подключение. Это делается путем установки поля `externalTrafficPolicy` в секции `spec` ресурса службы:

```
spec:
  externalTrafficPolicy: Local
  ...
```

Если определение службы содержит этот параметр и внешнее подключение открыто через порт узла службы, служебный прокси выберет локально работающий модуль. Если локальных модулей не существует, подключение зависнет (оно не будет перенаправлено в случайный глобальный модуль, именно так подключения обрабатываются, когда аннотация не используется). Поэтому необходимо обеспечить, чтобы подсистема балансировки нагрузки перенаправляла подключения только к тем узлам, которые имеют, по крайней мере, один такой модуль.

Использование этой аннотации имеет и другие недостатки. Обычно подключения распределяются равномерно по всем модулям, но при использовании упомянутой аннотации это уже не так.

Представьте себе два узла и три модуля. Предположим, узел A управляет одним модулем, а узел B – двумя другими. Если балансировщик нагрузки равномерно распределяет подключения между двумя узлами, то модуль на узле A получит 50% всех подключений, а два модуля на узле B получат всего 25% каждый, как показано на рис. 5.8.

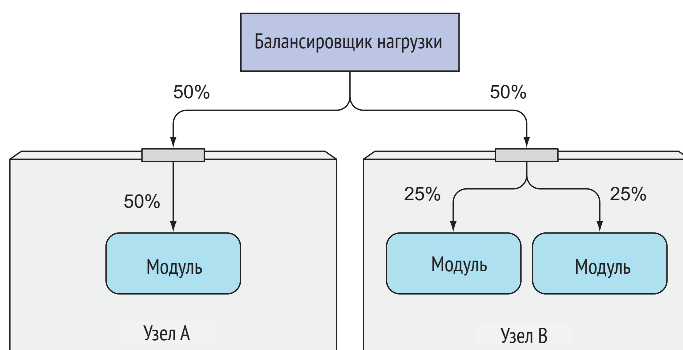


Рис. 5.8. Служба, использующая политику внешнего трафика Local, может привести к неравномерному распределению нагрузки между модулями

Изменчивость клиентского IP-адреса

Обычно, когда клиенты внутри кластера подключаются к службе, модули, с которыми эта служба связана, могут получать IP-адрес клиента. Но когда подключение получено через порт узла, исходный IP-адрес пакетов изменя-

ется, потому что на пакетах выполняется преобразование исходных сетевых адресов (Source Network Address Translation, SNAT).

Модуль не может видеть IP-адрес фактического клиента, что может быть проблемой для некоторых приложений, которым нужно знать клиентский IP-адрес. В случае веб-сервера, например, это означает, что журнал доступа не будет показывать IP-адрес браузера.

Политика внешнего трафика `Local`, описанная в предыдущем разделе, влияет на сохранение исходного IP-адреса, так как между узлом, принимающим подключение, и узлом, на котором размещен целевой модуль, нет дополнительного перехода (SNAT не выполняется).

5.4 Обеспечение доступа к службам извне через ресурс Ingress

Теперь вы увидели два способа обеспечения доступа к службе клиентам за пределами кластера, но существует еще один метод – создание ресурса Ingress.

> **ОПРЕДЕЛЕНИЕ.** *Ingress* (существительное) – это вход или вхождение; право на вход; средство или место входа; проход.

Позвольте мне сначала объяснить, почему вам нужен еще один способ доступа к службам Kubernetes извне.

Зачем нужны Ingress'ы

Одна из важных причин заключается в том, что для каждой службы `LoadBalancer` требуется собственный балансировщик нагрузки с собственным общедоступным IP-адресом, в то время как для Ingress'а требуется только один, даже когда предоставляется доступ к десяткам служб. Когда клиент отправляет HTTP-запрос ко входу, хост и путь в запросе определяют, к какой службе этот запрос перенаправляется. Это показано на рис. 5.9.

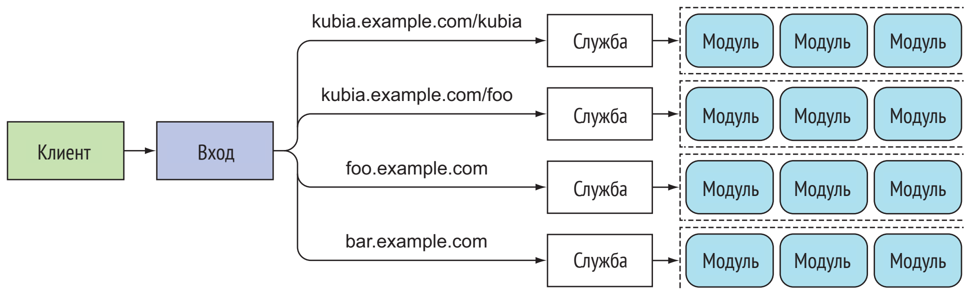


Рис. 5.9. Доступ к нескольким службам может быть обеспечен через один вход

Ingress'ы оперируют на уровне приложений сетевого стека (HTTP) и могут предоставлять такой функционал, как сохранение сессий на основе файлов cookie и т. п., чего не могут делать службы.

Для чего требуется Ingress

Прежде чем мы перейдем к функциональности, которую предоставляет объект Ingress, следует подчеркнуть, что, для того чтобы заставить ресурсы Ingress работать, в кластере должен быть запущен контроллер Ingress. Различные среды Kubernetes используют различные реализации данного контроллера, а некоторые вообще не предоставляют контроллер по умолчанию.

Например, Google Kubernetes Engine использует собственные средства балансировки нагрузки HTTP, которые обеспечивают облачную платформу Google Cloud Platform функциональностью Ingress. Первоначально Minikube не предоставлял контроллер в готовом виде, но теперь он включает в себя надстройку, которая может быть активирована, чтобы позволить вам опробовать функциональность объекта Ingress. Следуйте инструкциям в следующей ниже вставке, чтобы убедиться, что она включена.

Включение функционала Ingress в Minikube

Если для запуска примеров в этой книге вы используете Minikube, то вам нужно убедиться, что надстройка Ingress активирована. Это можно проверить, выведя список всех надстроек:

```
$ minikube addons list
- default-storageclass: enabled
- kube-dns: enabled
- heapster: disabled
- ingress: disabled              <- Надстройка Ingress не активирована
- registry-creds: disabled
- addon-manager: enabled
- dashboard: enabled
```

Вы узнаете о том, что из себя представляют эти надстройки, далее в этой книге, но к настоящему моменту вам должно быть совершенно очевидно, что такое надстройки `dashboard` и `kube-dns`. Активируйте надстройку Ingress, чтобы вы могли увидеть Ingress'ы в действии:

```
$ minikube addons enable ingress
ingress was successfully enabled
```

Эта команда должна была развернуть контроллер Ingress как еще один модуль. Скорее всего, модуль контроллера будет в пространстве имен `kube-system`, но не обязательно, поэтому с помощью параметра `--all-namespaces` выведите список всех запущенных модулей во всех пространствах имен:

```
$ kubectl get po --all-namespaces
NAMESPACE    NAME                              READY  STATUS   RESTARTS  AGE
default      kubia-rsv5m                       1/1    Running  0         13h
default      kubia-fe4ad                       1/1    Running  0         13h
```

(Продолжение)

```
default      kubia-ke823                       1/1    Running  0         13h
kube-system  default-http-backend-5wb0h        1/1    Running  0         18m
kube-system  kube-addon-manager-minikube       1/1    Running  3         6d
kube-system  kube-dns-v20-101vq                3/3    Running  9         6d
kube-system  kubernetes-dashboard-jxd9l        1/1    Running  3         6d
kube-system  nginx-ingress-controller-gdts0    1/1    Running  0         18m
```

В нижней части результата вы видите модуль контроллера Ingress. Имя предполагает, что для обеспечения функциональности Ingress используется Nginx (HTTP-сервер с открытым исходным кодом и обратным прокси).

СОВЕТ. Параметр `--all-namespaces`, упомянутый во вставке, удобен, если вы не знаете, в каком пространстве имен находится ваш модуль (или другой тип ресурса) либо если вы хотите вывести список ресурсов во всех пространствах имен.

5.4.1 Создание ресурса Ingress

Вы убедились, что в вашем кластере работает контроллер Ingress, поэтому теперь вы можете создать ресурс Ingress. Следующий ниже листинг показывает, как выглядит манифест YAML для ресурса Ingress.

Листинг 5.13. Определение ресурса Ingress: kubia-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
  - host: kubia.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kubia-nodeport
          servicePort: 80
```

Этот Ingress увязывает доменное имя kubia.example.com с вашей службой

Все запросы будут отправляться на порт 80 службы kubia-nodeport

Это определение задает ресурс Ingress с единственным правилом, которое удостоверяется, что все HTTP-запросы, полученные контроллером входа, в котором запрашивает хост `kubia.example.com`, будут отправлены в службу `kubia-nodeport` на порту 80.

ПРИМЕЧАНИЕ. От контроллеров входа на облачных провайдерах (например, в GKE) требуется, чтобы Ingress указывал на службу `NodePort`. Но это не является требованием самого Kubernetes.

5.4.2 Доступ к службе через Ingress

Для того чтобы получить доступ к службе через http://kubia.example.com, вы должны убедиться, что доменное имя привязано к IP-адресу контроллера Ingress.

Получение IP-адреса входа

Для того чтобы взглянуть на IP-адрес, вам нужно вывести список Ingress'ов:

```
$ kubectl get ingresses
NAME    HOSTS               ADDRESS          PORTS   AGE
kubia   kubia.example.com   192.168.99.100   80      29m
```

IP-адрес показан в столбце ADDRESS.

> **ПРИМЕЧАНИЕ.** При работе с поставщиками облачных служб, для того чтобы появились адреса, может уйти некоторое время, потому что за кадром контроллер Ingress запускает балансировщик нагрузки.

Обеспечение соответствия имени хоста, назначенного в Ingress, IP-адресу Ingress

После того как вы узнали IP-адрес, можно настроить DNS-серверы для привязки kubia.example.com к этому IP-адресу либо добавить в /etc/hosts (или в C:\windows\system32\drivers\etc\hosts в Windows) следующую строку:

```
192.168.99.100 kubia.example.com
```

Доступ к модулям через Ingress

Теперь все настроено, поэтому доступ к службе можно получить по адресу http://kubia.example.com (используя браузер или curl):

```
$ curl http://kubia.example.com
You've hit kubia-ke823
```

Вы успешно получили доступ к службе через Ingress. Давайте получше рассмотрим, как это происходило.

Как работает Ingress

На рис. 5.10 показано, как клиент подключился к одному из модулей через контроллер Ingress. Клиент сначала выполнил DNS-поиск kubia.example.com, и DNS-сервер (или локальная операционная система) вернул IP-адрес контроллера Ingress. Затем клиент отправил HTTP-запрос контроллеру Ingress и в заголовке Host указал kubia.example.com. Из этого заголовка контроллер определил, к какой службе клиент пытается обратиться, отыскал IP-адреса модулей через связанный со службой объект Endpoints и перенаправил запрос клиента одному из модулей.

Как вы можете видеть, контроллер Ingress не перенаправил запрос в службу. Он использовал ее только для выбора модуля. Большинство, если не все, контроллеров работает именно так.

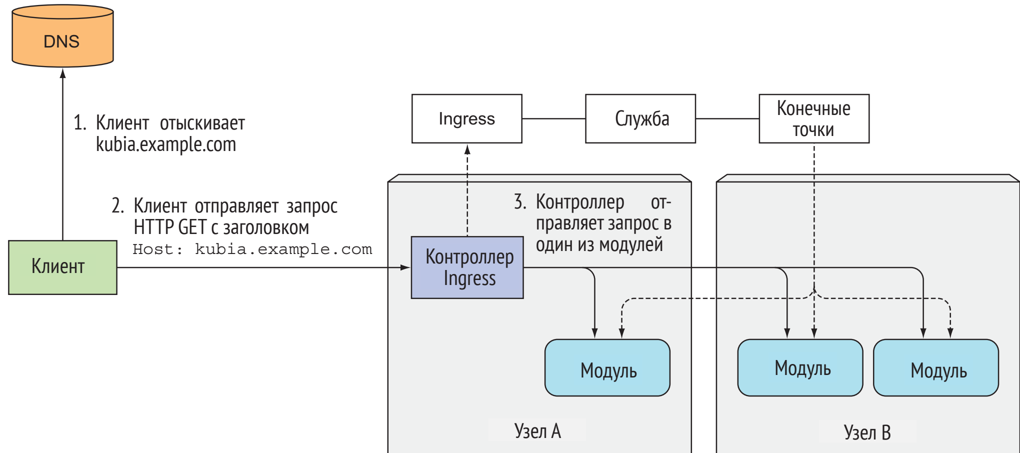


Рис. 5.10. Доступ к модулям через Ingress

5.4.3 Обеспечение доступа ко множеству служб через один и тот же Ingress

Если вы внимательно посмотрите на секцию `spec` ресурса Ingress, то увидите, что и `rules`, и `paths` являются массивами, поэтому они могут содержать несколько элементов. Как вы увидите далее, ресурс Ingress может увязывать несколько хостов и путей с несколькими службами. Давайте сначала сосредоточимся на путях `paths`.

Увязывание разных служб с разными путями одного хоста

Как показано в следующем ниже листинге, вы можете увязать несколько путей `paths` на одном узле с разными службами.

Листинг 5.14. Ingress, обеспечивающий доступ ко множеству служб на одном хосте, но с разными путями paths

```
...
  - host: kubia.example.com
    http:
      paths:
      - path: /kubia
        backend:
          serviceName: kubia
          servicePort: 80
```

Запросы к kubia.example.com/kubia будут перенаправлены в службу kubia

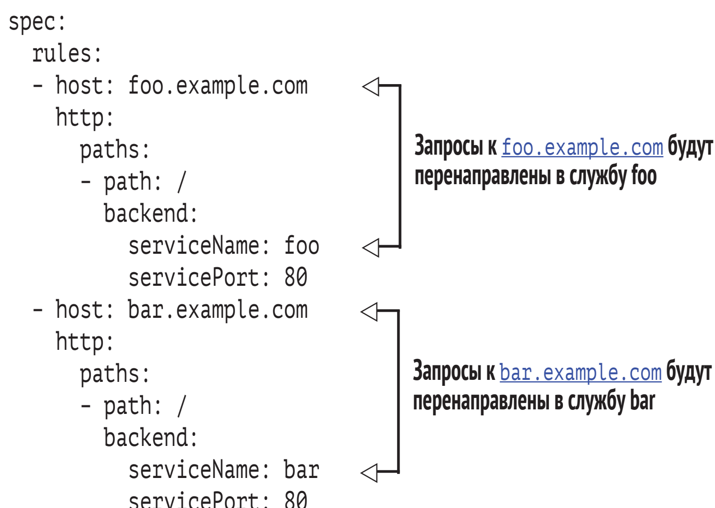
```
    - path: /foo
      backend:
        serviceName: bar
        servicePort: 80
```


В данном случае запросы будут отправлены в две разные службы в зависимости от пути в запрашиваемом URL-адресе. Поэтому клиенты могут достичь 2 разных служб через единый IP-адрес (адрес контроллера Ingress).

Увязывание разных служб с разными узлами

Аналогичным образом Ingress можно использовать для увязывания разных служб на основе хоста в HTTP-запросе, а не (только) пути, как показано в следующем далее листинге.

Листинг 5.15. Ingress, обеспечивающий доступ ко множеству служб на разных хостах



```
spec:
  rules:
  - host: foo.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: foo
          servicePort: 80
  - host: bar.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: bar
          servicePort: 80
```


Запросы, полученные контроллером, будут переадресованы либо в службу `foo`, либо службу `bar`, в зависимости от заголовка `Host` в запросе (так обрабатываются виртуальные хосты в веб-серверах). DNS-записи как для `foo.example.com`, так и для `bar.example.com` должны указывать на IP-адрес Ingress.

5.4.4 Настройка входа для обработки трафика TLS

Вы видели, как Ingress перенаправляет HTTP-трафик. Но как насчет HTTPS? Давайте кратко рассмотрим, как настроить Ingress для поддержки TLS.

Создание сертификата TLS для Ingress

Когда клиент открывает подключение TLS к контроллеру Ingress, контроллер терминирует соединение TLS. Коммуникация между клиентом и контроллером шифруется, тогда как коммуникация между контроллером и бэкенд-мо-

дулем – нет. Работающее в модуле приложение не нуждается в поддержке TLS. Например, если в модуле запущен веб-сервер, то он может принимать только HTTP-трафик и давать контроллеру Ingress заботиться обо всем, что связано с TLS. Для того чтобы позволить данному контроллеру это делать, необходимо прикрепить к Ingress сертификат и закрытый ключ. Они должны храниться в ресурсе Kubernetes под названием Secret, на который затем ссылается манифест ресурса Ingress. Мы подробно объясним секреты в главе 7. На данный момент вы создадите секрет, не уделяя ему слишком много внимания.

Сначала вам нужно создать закрытый ключ и сертификат:

```
$ openssl genrsa -out tls.key 2048
$ openssl req -new -x509 -key tls.key -out tls.cert -days 360 -subj
➥  /CN=kubia.example.com
```

Затем вы создаете секрет из двух файлов. Это делается следующим образом:

```
$ kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
secret "tls-secret" created
```

Подписание сертификатов через ресурс CertificateSigningRequest

Вместо того чтобы подписывать сертификат самостоятельно, вы можете получить сертификат, подписанный путем создания ресурса `CertificateSigningRequest` (CSR). Пользователи или их приложения могут создавать обычный запрос сертификата, помещать его в CSR, а затем либо оператор, либо автоматизированный процесс может подтверждать запрос следующим образом:

```
$ kubectl certificate approve <name of the CSR>
```

Подписанный сертификат может потом быть извлечен из поля `status.certificate` CSR.

Обратите внимание, что в кластере должен работать компонент подписчика сертификата; в противном случае создание запроса `CertificateSigningRequest` и его утверждение или отклонение не будет иметь никакого эффекта.

Закрытый ключ и сертификат теперь хранятся в секрете `tls-secret`. Теперь можно обновить объект Ingress, чтобы он также принимал запросы HTTPS для `kubia.example.com`. Манифест ресурса Ingress теперь должен выглядеть следующим образом.

Листинг 5.16. Обработка входом трафика TLS: kubia-ingress-tls.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
```

```
spec:
  tls:                               Вся конфигурация TLS находится под этим атрибутом
  - hosts:
    - kubia.example.com              Подключения TLS будут приниматься
    secretName: tls-secret           для хост-имени kubia.example.com
  rules:
  - host: kubia.example.com          Закрытый ключ и сертификат должны
    http:                            быть получены из ранее созданного
      paths:                         секрета tls-secret
      - path: /
        backend:
          serviceName: kubia-nodeport
          servicePort: 80
```

СОВЕТ. Вместо удаления ресурса Ingress и его воссоздания из нового файла можно вызвать команду `kubectl apply -f kubia-ingress-tls.yaml`, которая обновляет ресурс Ingress тем, что указано в файле.

Теперь вы можете использовать HTTPS для доступа к своей службе через Ingress:

```
$ curl -k -v https://kubia.example.com/kubia
* About to connect() to kubia.example.com port 443 (#0)
...
* Server certificate:
* subject: CN=kubia.example.com
...
> GET /kubia HTTP/1.1
> ...
You've hit kubia-xueq1
```

Результат команды показывает отклик приложения, а также сертификат сервера, с помощью которым вы настроили вход.

ПРИМЕЧАНИЕ. Поддержка функциональных средств объекта Ingress зависит от различных реализаций контроллера Ingress, поэтому проверьте документацию по конкретной реализации, чтобы увидеть, что именно поддерживается.

Ingress'ы являются относительно новым функциональным средством системы Kubernetes, поэтому в будущем можно ожидать много улучшений и новых функциональных возможностей. Хотя в настоящее время они поддерживают только балансировку нагрузки L7 (HTTP/HTTPS), также планируется поддержка балансировки нагрузки L4.

5.5 Сигналы о готовности модуля к приему подключений

Относительно служб и входов мы должны рассмотреть еще один аспект. Вы уже узнали, что модули задействуются в качестве конечных точек службы, если их метки соответствуют селектору модулей службы. После того как был создан новый модуль с правильными метками, он будет частью службы, и запросы начинают перенаправляться в модуль. Но что делать, если модуль не готов немедленно начать обслуживать запросы?

Для загрузки конфигурации или данных модулю может потребоваться время, либо ему может потребоваться выполнить процедуру разогрева, чтобы первый запрос пользователя не занял слишком много времени и не повлиял на работу пользователя. В таких случаях вы не хотите, чтобы модуль немедленно начинал получать запросы, в особенности когда уже работающие экземпляры могут обрабатывать запросы правильно и быстро. Имеет смысл не перенаправлять запросы в модуль, который находится в процессе запуска, до тех пор, пока он не будет полностью готов.

5.5.1 Знакомство с проверкой готовности

В предыдущей главе вы познакомились с проверками живучести и узнали о том, как они помогают поддерживать работоспособность приложений, обеспечивая автоматический перезапуск нездоровых контейнеров. Аналогично проверкам живучести, Kubernetes также позволяет определять для вашего модуля проверку готовности.

Проверка готовности периодически вызывается и определяет, должен конкретный модуль получать запросы клиентов или нет. Когда проверка готовности контейнера возвращает успех, это является сигналом к тому, что контейнер готов принимать запросы.

Идея готовности, очевидно, является чем-то специфическим для каждого контейнера. Система Kubernetes может просто проверить, откликается работающее в контейнере приложение на простой `GET`-контейнер или нет, либо же она может попадать на определенный URL-адрес, который заставляет приложение выполнять весь список проверок, позволяющий определять его готовность. Такая детальная проверка готовности, которая учитывает специфику приложения, является ответственностью разработчика приложения.

Типы проверок готовности

Как и проверки живучести, существует три вида проверок готовности:

- *проверка Exec*, при которой исполняется процесс. Состояние контейнера определяется кодом состояния на выходе процесса;
- *проверка HTTP GET*, которая отправляет запрос HTTP `GET`-контейнеру, и код состояния HTTP-отклика определяет готовность контейнера;

- *проверка сокета TCP*, которая открывает TCP-подключение к указанному порту контейнера. Если подключение установлено, то контейнер считается готовым.

Работа проверок готовности

При запуске контейнера система Kubernetes может быть настроена на ожидание заданного периода времени перед выполнением первой проверки готовности. После этого она периодически вызывает проверку и действует на основании результата проверки готовности. Если модуль сообщает, что он не готов, он удаляется из службы. Если модуль потом становится готовым, он снова добавляется.

В отличие от проверок живучести, если контейнер не проходит проверку готовности, он не убивается и не перезапускается. В этом состоит важное различие между проверками живучести и готовности. Проверки живучести поддерживают модули здоровым, убивая нездоровые контейнеры и заменяя их новыми, здоровыми, тогда как проверки готовности гарантируют, что запросы получают только те модули, которые готовы их обслуживать. Это главным образом необходимо во время исходного запуска контейнера, но также полезно после того, как контейнер проработал некоторое время.

Как можно видеть по рис. 5.11, если проверка готовности модуля не срабатывает, то модуль удаляется из объекта Endpoints. Подключающиеся к службе клиенты в этот модуль перенаправляться не будут. Эффект такой же, как если бы модуль вообще не соответствовал селектору меток службы.

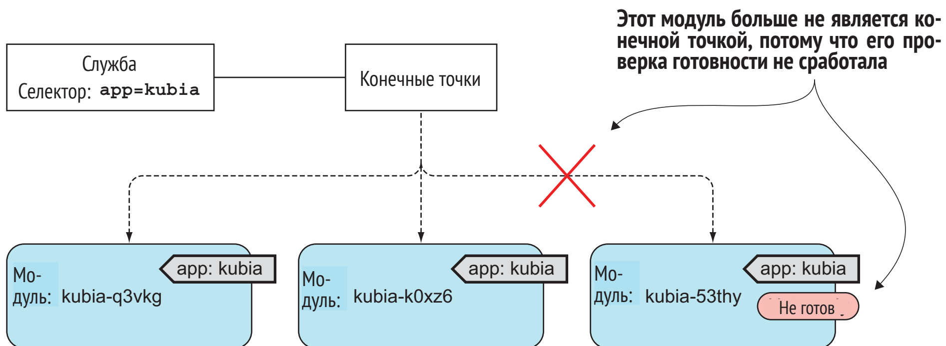


Рис. 5.11. Модуль, проверка готовности которого не срабатывает, удаляется из оконечной точки службы

В чем важность проверок готовности

Представьте, что группа модулей (например, модули, на которых работают серверы приложений) зависит от службы, предоставляемой другим модулем (например, бэкенд-базой данных). Если в какой-то момент один из фронтенд-модулей испытывает проблемы с подключением и больше не может достичь базы данных, может оказаться разумным, чтобы его проверка готовности отправила сигнал в Kubernetes, что модуль пока не готов обслуживать

любые запросы. Если другие экземпляры модуля не испытывают такого же типа проблем с подключением, они могут обслуживать запросы обычным образом. Проверка готовности удостоверяет, что клиенты коммуницируют только со здоровыми модулями и никогда не заметят, что с системой что-то не так.

5.5.2 Добавление в модуль проверки готовности

Далее вы добавите проверку готовности в свои существующие модули, изменив шаблон модуля контроллера репликации.

Добавление проверки готовности в шаблон модуля

Для того чтобы добавить проверку в шаблон модуля в вашем существующем контроллере репликации, вы будете использовать команду `kubectl edit`:

```
$ kubectl edit rc kubia
```

Когда YAML ресурса ReplicationController откроется в текстовом редакторе, найдите спецификацию контейнера в шаблоне модуля и добавьте следующее ниже определение проверки готовности в первый контейнер в секции `spec.template.spec.containers`. YAML должен выглядеть следующим образом.

Листинг 5.17. Контроллер репликации, создающий модуль с проверкой готовности: kubia-rc-readinessprobe.yaml

```
apiVersion: v1
kind: ReplicationController
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
        readinessProbe:
          exec:
            command:
            - ls
            - /var/ready
        ...
```

Проверка readinessProbe может быть определена для каждого контейнера в модуле

Проверка готовности будет периодически выполнять команду `ls /var/ready` внутри контейнера. Команда `ls` возвращает нулевой код выхода, если файл существует, либо ненулевой код выхода в противном случае. Если файл существует, то проверка готовности завершится успешно; в противном случае она не сработает.

Причина, по которой вы определяете такую странную проверку готовности, заключается в том, чтобы вы можете менять ее результат, создавая или удаляя указанный файл. Файл еще не существует, поэтому все модули должны теперь сообщить о неготовности, правильно? Не совсем. Как вы помните из предыдущей главы, изменение шаблона модуля контроллера репликации на существующие модули не влияет.

Другими словами, во всех ваших существующих модулях никакая проверка готовности по-прежнему не определена. Вы можете увидеть это, выведя список модулей с помощью команды `kubectl get pods` и взглянув на столбец `READY`. Вам нужно удалить модули и воссоздать их контроллером репликации. Новые модули не будут проходить проверку готовности и не будут включены в качестве конечных точек службы до тех пор, пока в каждом из них не будет создан файл `/var/ready`.

Наблюдение и модификация состояния готовности модулей

Еще раз выведите список модулей и проверьте их готовность:

```
$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
kubia-2r1qb   0/1     Running   0          1m
kubia-3rax1   0/1     Running   0          1m
kubia-3yw4s   0/1     Running   0          1m
```

Столбец `READY` показывает, что ни один из контейнеров не готов. Теперь сделайте так, чтобы проверка готовности одного из них начала возвращать успех, создав файл `/var/ready`, существование которого делает вашу фиктивную проверку готовности успешной:

```
$ kubectl exec kubia-2r1qb -- touch /var/ready
```

Вы использовали команду `kubectl exec` для выполнения команды `touch` внутри контейнера модуля `kubia-2r1qb`. Команда `touch` создает файл, если он еще не существует. Команда проверки готовности модуля должна теперь выйти с кодом состояния 0, то есть проверка была успешной, и модуль должен теперь быть показан как готовый. Давайте посмотрим, так ли это:

```
$ kubectl get po kubia-2r1qb
NAME          READY   STATUS    RESTARTS   AGE
kubia-2r1qb   0/1     Running   0          2m
```

Модуль по-прежнему не готов. Что-то не так, или это ожидаемый результат? Приглядимся поближе к модулю с помощью команды `kubectl describe`. Результат должен содержать следующую ниже строку:

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1
➥ #failure=3
```

Проверка готовности выполняется периодически – по умолчанию каждые 10 секунд. Модуль не готов, потому что проверка готовности еще не была вызвана. Но не позднее чем через 10 секунд модуль должен быть готов, и его IP-адрес должен быть указан как единственная конечная точка службы (для подтверждения выполните команду `kubectl get endpoints kubia-loadbalancer`).

Попадание в службу с единственным готовым модулем

Теперь вы можете зайти на URL-адрес службы несколько раз, чтобы увидеть, что каждый запрос перенаправляется на этот модуль:

```
$ curl http://130.211.53.173
You've hit kubia-2r1qb
$ curl http://130.211.53.173
You've hit kubia-2r1qb
...
$ curl http://130.211.53.173
You've hit kubia-2r1qb
```

Даже если будут работать три модуля, только один модуль сообщает о готовности, и поэтому получать запросы будет только один модуль. Если теперь удалить файл, то модуль снова будет удален из службы.

5.5.3 Что должны делать реальные проверки готовности

Такая фиктивная проверка готовности полезна только для демонстрации того, что вообще делают проверки готовности. В реальном мире проверка готовности должна возвращать успех или неуспех в зависимости от того, может приложение (и хочет) получать клиентские запросы или нет.

Ручное удаление модулей из служб должно выполняться либо удалением модуля, либо изменением меток модуля, а не ручной переустановкой переключателя в проверке.

> **СОВЕТ.** Если вы хотите добавить или удалить модуль из службы вручную, добавьте в модуль и в селектор меток службы метку `enabled=true`. Удалите эту метку, если необходимо удалить модуль из службы.

Всегда задавайте проверку готовности

Прежде чем мы завершим этот раздел, я должен подчеркнуть два последних аспекта относительно проверок готовности. Прежде всего, если вы не добавите проверку готовности в свои модули, они почти сразу станут конечными точками обслуживания. Если вашему приложению требуется слишком много времени, для того чтобы начать прослушивать входящие подключения, клиентские запросы, поступающие в службу, будут перенаправляться в модуль, в то время когда он все еще запускается, и не будут готовы принимать входящие подключения. Поэтому клиенты будут видеть такие ошибки, как «Connection refused».

СОВЕТ. Следует всегда определять проверку готовности, даже если она настолько проста, что сводится к отправке HTTP-запроса на базовый URL.

Не включайте в проверки готовности логику отключения модуля

Другой аспект, который я бы хотел отметить, относится к иному концу жизни модуля (завершению работы модуля), а он также связан с клиентами, которые испытывают ошибки подключения.

Когда модуль завершает работу, работающее в нем приложение обычно прекращает принимать подключения, как только оно получает сигнал к завершению. По этой причине вы, возможно, думаете, что вам нужно заставить проверку готовности начать не срабатывать, как только инициируется процедура завершения работы, тем самым гарантируя, что модуль будет удален из всех служб, в которые он входит. Но это не обязательно делать, потому что как только вы удаляете модуль, Kubernetes удаляет модуль из всех служб.

5.6 Использование служб без обозначенной точки входа (Headless-сервисов) для обнаружения индивидуальных модулей

Вы видели, каким образом службы могут использоваться для обеспечения стабильного IP-адреса, позволяя клиентам подключаться к модулям (или другим конечным точкам), подключенным к каждой такой службе. Каждое подключение к службе перенаправляется на один случайно выбранный поддерживающий модуль. Однако что делать, если клиенту нужно подключиться ко всем этим модулям? Что делать, если самим таким модулям нужно подключиться ко всем другим модулям? Подключение через службу – явно не тот способ, чтобы это сделать. А что тогда?

Для того чтобы клиенту подключиться ко всем модулям, требуется выяснить IP-адрес каждого отдельного модуля. Один из вариантов – дать клиенту обратиться к серверу Kubernetes API и получить список модулей и их IP-адресов с помощью вызовов API, но поскольку вы всегда должны стремиться держать свои приложения платформенно-независимыми от Kubernetes, то использование сервера API не подойдет.

К счастью, Kubernetes позволяет клиентам обнаруживать IP-адреса модулей посредством поиска в DNS. Обычно, когда вы выполняете DNS-запрос, DNS-сервер возвращает единственный кластерный IP-адрес службы. Но если вы сообщите системе Kubernetes, что для вашей службы вам не нужен кластерный IP-адрес (это можно сделать, присвоив полю `clusterIP` значение `None` в спецификации службы), то вместо единственного IP-адреса службы DNS-сервер будет возвращать IP-адреса модулей.

Вместо того чтобы возвращать одну A-запись DNS, DNS-сервер будет возвращать для службы несколько A-записей, каждая с указанием на IP-адрес отдельного модуля, поддерживающего службу в данный момент. Поэтому

клиенты могут выполнять простой запрос A-записи и получать IP-адреса всех модулей, входящих в состав службы. Затем клиент может использовать эту информацию для подключения к одному, нескольким или ко всем из них.

5.6.1 Создание службы без обозначенной точки входа

Присвоение полю `clusterIP` в спецификации службы значения `None` делает службу службой без обозначенной точки входа (Headless), так как Kubernetes не назначит ей кластерный IP-адрес, через который клиенты могли бы подключаться к поддерживающим ее модулям.

Сейчас вы создадите Headless-службу под названием `kubia-headless`. В следующем ниже листинге показано ее определение.

Листинг 5.18. Безголовая служба: kubia-svc-headless.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-headless
spec:
  clusterIP: None        <-- Это делает службу безголовой
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

После создания службы с помощью `kubectl create` вы можете проверить ее командами `kubectl get` и `kubectl describe`. Вы увидите, что у нее нет кластерного IP и ее конечные точки содержат модули (их часть), совпадающие с ее селектором модулей. Я говорю «часть», потому что ваши модули содержат проверку готовности, поэтому только готовые модули будут внесены в список как конечные точки службы. Прежде чем продолжить, убедитесь, что, по крайней мере, два модуля сообщают о готовности, создав файл `/var/ready`, как в предыдущем примере:

```
$ kubectl exec <pod name> -- touch /var/ready
```

5.6.2 Обнаружение модулей через DNS

Теперь, когда ваши модули готовы, вы можете попробовать выполнить поиск в DNS, чтобы узнать, получаете ли вы фактические IP-адреса модулей или нет. Вам нужно будет выполнить поиск изнутри одного из модулей. К сожалению, образ контейнера `kubia` не содержит двоичный файл `nslookup` (или `dig`), поэтому его нельзя использовать для поиска в DNS.

Вы пытаетесь лишь выполнить поиск в DNS изнутри модуля, работающего в кластере. Почему бы не запустить новый модуль на основе образа, содержа-

щего необходимые двоичные файлы? Для выполнения действий, связанных с DNS, можно использовать образ контейнера `tutum/dnsutils`, который доступен в Docker Hub и содержит исполняемые файлы `nslookup` и `dig`. Для того чтобы запустить модуль, вы можете пройти весь процесс создания для него манифеста YAML и передачи его команде `kubectl create`, но такая работа – слишком долгая, не правда ли? К счастью, есть способ побыстрее.

Запуск модуля без написания манифеста YAML

В главе 1 вы уже создавали модули без написания манифеста YAML с помощью команды `kubectl run`. Но на этот раз вам требуется создать только модуль – вам не нужно создавать контроллер репликации для управления модулем. Это можно сделать следующим образом:

```
$ kubectl run dnsutils --image=tutum/dnsutils --generator=run-pod/v1
➥  --command -- sleep infinity
pod "dnsutils" created
```

Весь трюк заключается в параметре `--generator=run-pod / v1`, который поручает команде `kubectl` создать модуль напрямую, без какого-либо контроллера репликации или аналогичного ему.

Устройство A-записей DNS, возвращаемых для службы без обозначенной точки входа

Давайте воспользуемся вновь созданным модулем для выполнения поиска в DNS:

```
$ kubectl exec dnsutils nslookup kubia-headless
...
Name: kubia-headless.default.svc.cluster.local
Address: 10.108.1.4
Name: kubia-headless.default.svc.cluster.local
Address: 10.108.2.5
```

DNS-сервер возвращает два разных IP-адреса для FQDN `kubia-headless.default.svc.cluster.local`. Это IP-адреса двух модулей, которые сообщают о готовности. Вы можете подтвердить это, выведя список модулей с помощью команды `kubectl get pods -o wide`, которая показывает IP-адреса модулей.

Это отличается от того, что возвращает DNS для обычных (не Headless) служб, таких как ваша служба `kubia`, в которой возвращается кластерный IP-адрес службы:

```
$ kubectl exec dnsutils nslookup kubia
...
Name: kubia.default.svc.cluster.local
Address: 10.111.249.153
```

Хотя Headless-службы могут внешне отличаться от обычных служб, они не на столько уж отличаются с точки зрения клиентов. Даже с Headless-службой клиенты могут подключаться к ее модулям, связываясь с DNS-именем службы, как с обычными службами. Но поскольку DNS возвращает IP-адреса модулей, с Headless-службами клиенты подключаются непосредственно к модулям, а не через служебный прокси.

> **ПРИМЕЧАНИЕ.** Безголовая служба по-прежнему обеспечивает балансировку нагрузки между модулями, но только через механизм циклической обработки DNS, а не через служебный прокси.

5.6.3 Обнаружение всех модулей – даже тех, которые не готовы

Вы видели, что только готовые модули становятся конечными точками служб. Но иногда требуется использовать механизм обнаружения служб, который находит все модули, соответствующие селектору меток службы, даже те, которые не готовы.

К счастью, вам не придется обращаться к серверу API Kubernetes. Можно использовать механизм поиска в DNS, который находит даже неготовые модули. Для того чтобы сообщить Kubernetes, что в службу должны быть добавлены все модули, независимо от состояния готовности модуля, необходимо добавить в службу следующую аннотацию:

```
kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
```

> **ПРЕДУПРЕЖДЕНИЕ.** Как следует из названия аннотации, в момент написания этой книги данный функционал находился в версии альфа. API служб Kubernetes уже поддерживает новое поле секции `spec` под названием `publishNotReadyAddresses`, которое заменит аннотацию `tolerate-unready-endpoints`. В Kubernetes версии 1.9.0 это поле еще не учтено (именно аннотация определяет, включаются неготовые конечные точки в DNS или нет). Проверьте документацию, чтобы увидеть, изменилась ли эта ситуация.

5.7 Устранение неполадок в службах

Службы являются важной концепцией Kubernetes и источником разочарования для многих разработчиков. Я видел, как многие разработчики теряют кучу времени, выясняя, почему они не могут подключиться к своим модулям через IP-адрес службы или полное доменное имя. По этой причине требуется провести краткий обзор того, как устранять неполадки служб.

Если вы не можете получить доступ к своим модулям через службу, то вам следует начать со следующего списка:

- прежде всего убедитесь, что вы подключаетесь к кластерному IP-адресу службы изнутри кластера, а не извне;
- не тратьте времени на пингование IP-адреса службы, чтобы выяснить, доступна ли служба (напомню, что кластерный IP-адрес службы – это виртуальный IP-адрес, и его пингование не сработает);
- если вы определили проверку готовности, убедитесь, что она выполняется успешно; в противном случае модуль не будет частью службы;
- для того чтобы убедиться, что модуль является частью службы, проинспектируйте соответствующий объект Endpoints с помощью команды `kubectl get endpoints`;
- если вы пытаетесь получить доступ к службе через ее полное доменное имя или его часть (например, `myservice.mynamespace.svc.cluster.local` или `myservice.mynamespace`) и это не работает, посмотрите, сможете ли вы обратиться к нему, используя ее кластерный IP-адрес вместо полного доменного имени;
- проверьте, подключаетесь ли вы к предоставляемому службой порту, а не к целевому порту;
- попробуйте подключиться к IP-адресу модуля напрямую, чтобы подтвердить, что ваш модуль принимает подключения на правильном порту;
- если вы не можете получить доступ к своему приложению даже через IP-адрес модуля, убедитесь, что ваше приложение не открыто своими портами только к localhost-адресу.

Эта краткая инструкция поможет вам решить большинство проблем, связанных со службами. Подробнее о том, как работают службы, вы узнаете в главе 11. Точное понимание того, как они реализованы, должно во многом упростить устранение в них неисправностей.

5.8 Резюме

В этой главе вы узнали, как создавать ресурсы Service системы Kubernetes для предоставления доступа к службам, имеющимся в вашем приложении, независимо от того, сколько экземпляров модулей предоставляют каждую службу. Вы узнали, как Kubernetes:

- обеспечивает доступ ко множеству модулей, которые отождествляются с некоторым селектором меток под единым, стабильным IP-адресом и портом;
- делает службы доступными по умолчанию изнутри кластера, но позволяет сделать службу доступной за пределами кластера, установив для нее тип `NodePort` или `LoadBalancer`;
- позволяет модулям обнаруживать службы вместе с их IP-адресами и портами путем поиска в переменных среды;

- позволяет обнаруживать службы, расположенные за пределами кластера, и обмениваться с ними информацией, создавая ресурс Service без указания селектора, путем создания ассоциированного ресурса Endpoints;
- предоставляет DNS-псевдоним `CNAME` для внешних служб с типом `ExternalName`;
- обеспечивает доступ к нескольким службам HTTP через один вход Ingress (потребляя один IP-адрес);
- использует проверку готовности контейнера модуля для определения, должен модуль быть включен в качестве конечной точки службы или нет;
- активирует обнаружение IP-адресов модулей через DNS при создании безголовой службы.

Наряду с получением более глубокого понимания служб вы также узнали, как:

- устранять неполадки в службах;
- изменять правила брандмауэра в Google Kubernetes / Compute Engine;
- выполнять команды в контейнерах модулей посредством `kubectl exec`;
- выполнять оболочку `bash` в контейнере существующего модуля;
- модифицировать ресурсы Kubernetes посредством команды `kubectl apply`;
- запускать неуправляемый нерегламентированный модуль с помощью команды `kubectl run --generator=run-pod/v1`.

Глава 6

Тома: подключение дискового хранилища к контейнерам

Эта глава посвящена:

- созданию многоконтейнерных модулей;
- созданию тома для общего использования дискового хранилища между контейнерами;
- использованию репозитория Git внутри модуля;
- прикреплению к модулям постоянного хранилища, например постоянного хранилища GCE Persistent Disk;
- использованию предварительно зарезервированного постоянного хранилища;
- динамическому резервированию постоянного хранилища.

В предыдущих трех главах мы познакомились с модулями и другими ресурсами Kubernetes, которые с ними взаимодействуют, а именно ресурсами ReplicationController, ReplicaSet, DaemonSet, Job и Service. Теперь мы заглянем внутрь модуля, чтобы узнать, каким образом его контейнеры получают доступ к внешнему дисковому хранилищу и/или делят между собой хранилище.

Мы отметили, что модули похожи на логические хосты, где процессы, работающие внутри них, делят между собой ресурсы, такие как ЦП, ОЗУ, сетевые интерфейсы и др. Можно было бы ожидать, что процессы также будут совместно использовать диски, но это не так. Вы помните, что каждый контейнер в модуле имеет свою собственную изолированную файловую систему, так как файловая система поступает из образа контейнера.

Каждый новый контейнер начинается с точного набора файлов, которые были добавлены в образ во время сборки. Добавьте к этому тот факт, что контейнеры в модуле перезапускаются (либо потому, что процесс умер, либо потому, что проверка живучести сигнализирует системе Kubernetes, что контейнер не здоров), и вы поймете, что новый контейнер не будет видеть ничего,

что было сохранено в файловой системе предыдущим контейнером, несмотря на то что по-новому запущенный контейнер выполняется в том же модуле.

В определенных сценариях требуется, чтобы новый контейнер продолжал работать с той же точки, где остановился последний, как это происходит, например, во время перезапуска процесса на физической машине. Вам, возможно, не понадобится (или вы не захотите), чтобы вся файловая система была сохранена; вам лишь нужно сохранить каталоги, которые содержат фактические данные.

Kubernetes обеспечивает это путем определения *томов* хранения. Они не являются ресурсами верхнего уровня, такими как модули, но вместо этого определяются как часть модуля и имеют тот же жизненный цикл, что и модуль. Это означает, что том создается при запуске модуля и уничтожается при удалении модуля. По этой причине содержимое тома будет сохраняться при перезапуске контейнера. После перезапуска контейнера новый контейнер может видеть все файлы, записанные в том предыдущим контейнером. Кроме того, если модуль содержит несколько контейнеров, то том может использоваться всеми из них одновременно.

6.1 Знакомство с томами

Тома Kubernetes являются компонентом модуля и, следовательно, определяются в спецификации модуля, подобно контейнерам. Они не являются автономным объектом Kubernetes и не могут быть созданы или удалены самостоятельно. Том доступен для всех контейнеров в модуле, но он должен быть установлен в каждом контейнере, который должен получать к нему доступ. В каждом контейнере можно смонтировать том в любом месте его файловой системы.

6.1.1 Объяснение томов на примере

Представьте, у вас есть модуль с тремя контейнерами (см. на рис. 6.1). Один контейнер управляет веб-сервером, который обслуживает HTML-страницы из каталога `/var/htdocs` и сохраняет журнал доступа в `/var/logs`. Второй контейнер управляет агентом, который создает HTML-файлы и сохраняет их в `/var/html`. Третий контейнер обрабатывает журналы, которые он находит в каталоге `/var/logs` (ротирует их, сжимает, анализирует или что-то еще).

Каждый контейнер имеет четко определенную ответственность, но сам по себе каждый контейнер не будет иметь большого смысла. Создание модуля с этими тремя контейнерами без совместного использования дискового хранилища не имеет никакого смысла, потому что генератор контента будет писать созданные HTML-файлы внутри собственного контейнера, и веб-сервер не сможет получать доступ к этим файлам, так как он работает в отдельном изолированном контейнере. Вместо этого он будет обслуживать пустой каталог или все, что вы помещаете в каталог `/var/htdocs` в образе его контейнера. Аналогичным образом ротатор журнала никогда ничего не сделает, потому

что его каталог `/var/logs` всегда будет оставаться пустым, ничего не записывая в журналы. Модуль с этими 3 контейнерами и отсутствующими томами, по существу, не делает ничего.

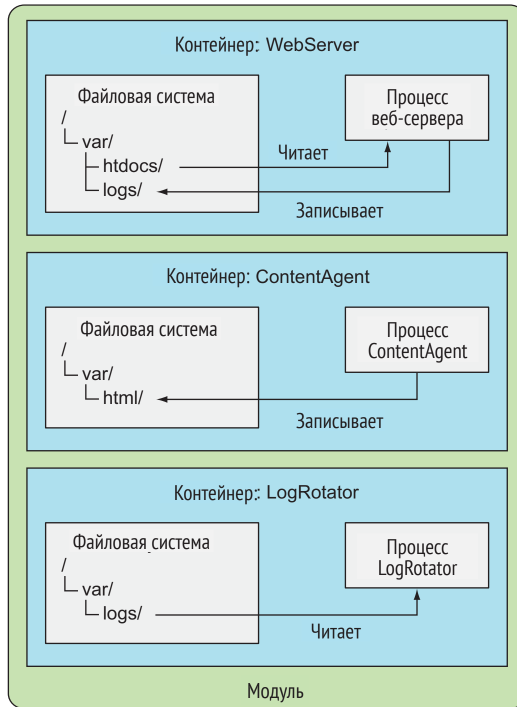


Рис. 6.1. Три контейнера одного модуля без общего хранилища

Но если вы добавите в модуль два тома и смонтируете их в соответствующих путях внутри трех контейнеров, как показано на рис. 6.2, то вы создадите систему, которая окажется намного больше, чем суммой ее частей. Linux позволяет монтировать файловую систему в дереве файлов в произвольных местах. При этом содержимое смонтированной файловой системы будет доступно в каталоге, в который она смонтирована. Смонтировав один и тот же том в два контейнера, они смогут работать с одними и теми же файлами. В вашем случае вы монтируете два тома в три контейнера. Сделав это, ваши три контейнера смогут работать вместе и делать что-то полезное. И давайте я объясню, как.

Прежде всего у модуля есть том `publicHtml`. Этот том монтируется в контейнере `WebServer` в каталоге `/var/htdocs`, так как это каталог, из которого веб-сервер обслуживает файлы. Тот же том также монтируется в контейнере `ContentAgent`, но в каталоге `/var/html`, так как именно в него агент записывает файлы. При таком монтировании одного тома веб-сервер теперь будет обслуживать содержимое, создаваемое агентом контента.

Аналогичным образом модуль также имеет том `logVol` для хранения журналов. Этот диск смонтирован в `/var/logs` в контейнерах `WebServer` и `LogRotator`. Обратите внимание, что он не монтируется в контейнер `ContentAgent`. Этот

контейнер не может обращаться к его файлам, даже если контейнер и том являются частью одного модуля. Определить в модуле том недостаточно; если вы хотите, чтобы контейнер имел к нему доступ, внутри спецификации контейнера необходимо также определить `VolumeMount`.

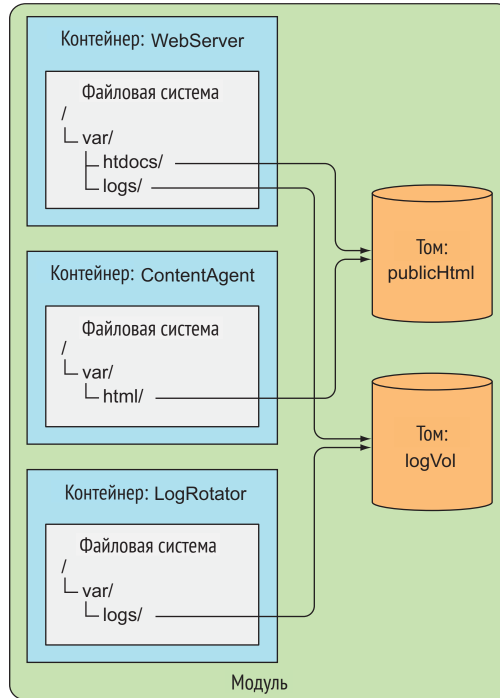


Рис. 6.2. Три контейнера с двумя томами, смонтированными в разных путях монтирования

Два тома в этом примере изначально могут быть пустыми, поэтому можно использовать тип тома `emptyDir`. Kubernetes также поддерживает другие типы томов, которые либо заполняются во время инициализации тома из внешнего источника, либо существующий каталог монтируется внутри тома. Этот процесс заполнения или монтирования тома выполняется до запуска контейнеров модуля.

Том привязан к жизненному циклу модуля и будет существовать только до тех пор, пока модуль существует, но в зависимости от типа тома файлы тома могут оставаться нетронутыми даже после того, как модуль и том исчезнут, а затем могут быть смонтированы в новый том. Давайте посмотрим, какие типы томов существуют.

6.1.2 Знакомство с типами томов

Имеется широкий диапазон типов томов. Некоторые из них являются универсальными, в то время как другие являются специфическими для фактических технологий хранения, используемых под ними. Не переживайте, если вы

никогда не слышали об этих технологиях, – я не слышал, по крайней мере, о половине из них. Вы, вероятно, будете использовать типы томов только для технологий, которые вы уже знаете и используете. Вот список нескольких из доступных типов томов:

- `emptyDir` – простой пустой каталог, используемый для хранения временных данных;
- `hostPath` – используется для монтирования каталогов из файловой системы рабочего узла в модуль;
- `gitRepo` – том, инициализируемый в ходе проверки содержимого репозитория Git;
- `nfs` – общий ресурс NFS, монтируемый в модуле;
- `gcePersistentDisk` (Google Compute Engine Persistent Disk), `awsElasticBlockStore` (Amazon Web Services Elastic Block Store Volume), `azureDisk` (Microsoft Azure Disk Volume) – используются для монтирования систем хранения данных, специфичных для поставщика облачных служб;
- `cinder`, `cephfs`, `iscsi`, `flocker`, `glusterfs`, `quobyte`, `rbd`, `flexVolume`, `vsphereVolume`, `photonPersistentDisk`, `scaleIO` – используются для монтирования других типов сетевых хранилищ;
- `configMap`, `secret`, `downwardAPI` – специальные типы томов, используемые для предоставления модулю определенных ресурсов Kubernetes и кластерной информации;
- `persistentVolumeClaim` – способ использовать заранее или динамически резервируемое постоянное хранилище. (О них мы поговорим в последнем разделе этой главы.)

Эти типы томов служат различным целям. О некоторых из них вы узнаете в следующих разделах. Специальные типы томов (`secret`, `downwardAPI`, `configMap`) рассматриваются в следующих двух главах, поскольку они используются не для хранения данных, а для предоставления метаданных системы Kubernetes приложениям, работающим в модуле.

Один модуль может одновременно использовать несколько томов разных типов, и, как мы уже упоминали ранее, в каждом контейнере модуля том может быть смонтирован или не смонтирован.

6.2 Использование томов для обмена данными между контейнерами

Хотя том может оказаться полезным, даже когда он используется одним контейнером, давайте сначала сосредоточимся на том, как он используется для обмена данными между несколькими контейнерами в модуле.

6.2.1 Использование тома emptyDir

Простейшим типом тома является том `emptyDir`, поэтому рассмотрим его в первом примере определения тома в модуле. Как следует из названия, этот

том начинается как пустой каталог. Приложение, запущенное внутри модуля, может записывать любые файлы, которые ему нужны. Поскольку время жизни этого тома связано со временем жизни модуля, содержимое тома теряется при удалении модуля.

Том `emptyDir` особенно полезен для обмена файлами между контейнерами, запущенными в одном модуле. Но он также может использоваться одним контейнером, когда контейнеру необходимо временно записать данные на диск, например при выполнении операции сортировки большого набора данных, который не может поместиться в доступную оперативную память. Данные также могут быть записаны в саму файловую систему контейнера (помните верхний слой чтения и записи в контейнере?), но между этими двумя вариантами существуют тонкие различия. Файловая система контейнера может быть недоступна для записи (мы поговорим об этом ближе к концу книги), поэтому запись на смонтированный том может быть единственным вариантом.

Использование тома emptyDir в модуле

Давайте вернемся к предыдущему примеру, где веб-сервер, агент контента и ротатор журнала делят между собой два тома, но давайте немного упростим. Вы создадите модуль, содержащий только контейнер веб-сервера и агент содержимого, а также один том для HTML.

В качестве веб-сервера вы будете использовать Nginx и для создания HTML-контента команду `fortune` системы Unix. Команда `fortune` печатает случайную цитату каждый раз, когда вы ее запускаете. Вы создадите скрипт, который будет вызывать команду `fortune` каждые 10 секунд и сохранять ее результат в файле index.html. Вы найдете существующий образ Nginx в Docker Hub, но вам нужно либо создать образ fortune самостоятельно, либо использовать тот, который я уже собрал и отправил в Docker Hub под `luksa/fortune`. Если вы хотите освежить сведения о способе создания образов Docker, обратитесь к вставке ниже.

Создание образа контейнера fortune

Вот как выполняется сборка образа. Создайте новый каталог под названием `fortune`, а затем внутри него создайте шелл-скрипт `fortuneloop.sh` со следующим содержимым:

```
#!/bin/bash
trap "exit" SIGINT
mkdir /var/htdocs
while :
do
  echo $(date) Writing fortune to /var/htdocs/index.html
  /usr/games/fortune > /var/htdocs/index.html
  sleep 10
done
```

(Продолжение)

Затем в том же каталоге создайте файл с именем `Dockerfile`, содержащий следующее:

```
FROM ubuntu:latest
RUN apt-get update ; apt-get -y install fortune
ADD fortuneloop.sh /bin/fortuneloop.sh
ENTRYPOINT /bin/fortuneloop.sh
```

Наш образ основан на образе `ubuntu:latest`, который по умолчанию не включает двоичный файл `fortune`. Вот почему во второй строке файла Dockerfile вы устанавливаете его с помощью `apt-get`. После этого вы добавляете сценарий `fortuneloop.sh` в папку `/bin` образа. В последней строке файла Dockerfile указывается, что сценарий `fortuneloop.sh` должен быть выполнен при запуске образа.

После подготовки обоих файлов создайте и закачайте образ в Docker Hub с помощью следующих двух команд (замените `luksa` на собственный ID пользователя в Docker Hub):

```
$ docker build -t luksa/fortune .
$ docker push luksa/fortune
```

Создание модуля

Теперь, когда у вас есть два образа, необходимых для запуска модуля, пришло время создать манифест модуля. Создайте файл под названием `fortune-pod.yaml` с содержимым, показанным в следующем ниже листинге.

Листинг 6.1. Модуль с двумя контейнерами, делящими между собой один том: fortune-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
  - image: luksa/fortune
    name: html-generator
    volumeMounts:
    - name: html
      mountPath: /var/htdocs
  - image: nginx:alpine
    name: web-server
```

Первый контейнер называется html-generator и запускает образ luksa/fortune

Том под названием html монтируется в /var/htdocs в контейнере

Второй контейнер называется web-server и запускает образ nginx:alpine

```
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    emptyDir: {}
```

Тот же том, что и выше, монтируется в /usr/share/nginx/html только для чтения

Единственный том emptyDir под названием html монтируется в два контейнера выше

Модуль содержит 2 контейнера и единственный том, который монтируется в оба из них, но в различных путях. При запуске контейнера `html-generator` он начинает каждые 10 секунд писать результат команды `fortune` в файл `/var/htdocs/index.html`. Поскольку том монтируется в `/var/htdocs`, файл `index.html` пишется в том, а не в верхний слой контейнера. Как только запускается контейнер `web-server`, он начинает обслуживать все HTML-файлы, находящиеся в каталоге `/usr/share/nginx/html` (каталог по умолчанию, из которого nginx обслуживает файлы). Поскольку вы смонтировали том именно в этом месте, nginx будет обслуживать файл `index.html`, записанный туда контейнером, выполняющим цикл с командой `fortune`. Конечным эффектом является то, что клиент, отправляющий HTTP-запрос в модуль на порту 80, в качестве отклика получит текущее сообщение с цитатой, полученное из `fortune`.

Обзор модуля в действии

Чтобы увидеть сообщение с цитатой, необходимо активировать доступ к модулю. Это можно сделать, перенаправив порт с локальной машины в модуль:

```
$ kubectl port-forward fortune 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

> **ПРИМЕЧАНИЕ.** В качестве упражнения вместо использования проброса портов вы также можете обеспечить доступ к модулю через службу.

Теперь вы можете получить доступ к серверу Nginx через порт 8080 вашей локальной машины. Для этого примените `curl`:

```
$ curl http://localhost:8080
Beware of a tall blond man with one black shoe.
```

Если подождать несколько секунд и отправить еще один запрос, то вы получите другое сообщение. Объединив два контейнера, вы создали простое приложение, для того чтобы увидеть то, как том может склеивать два контейнера и улучшать то, что каждый из них делает по отдельности.

Определение носителя для использования с томом emptyDir

Использованный вами том `emptyDir` был создан на фактическом диске рабочего узла, в котором размещен модуль, поэтому его производительность зависит от типа дисков узла. Но вы можете сообщить системе Kubernetes создать том `emptyDir` в файловой системе tmpfs (в памяти, а не на диске). Для этого присвойте свойству `medium` тома `emptyDir` значение `Memory`:

```
volumes:
  - name: html
    emptyDir:
      medium: Memory
```

Файлы этого тома emptyDir должны сохраняться в памяти

Том `emptyDir` является самым простым типом тома, но другие типы надстраиваются поверх него. После создания пустого каталога они заполняют его данными. Одним из таких типов тома является тип `gitRepo`, с которым мы познакомимся далее.

6.2.2 Использование репозитория Git в качестве отправной точки для тома

Том `gitRepo` – это, в сущности, том `emptyDir`, который заполняется путем клонирования репозитория Git и извлечения конкретной версии данных при запуске модуля (но до создания его контейнеров). На рис. 6.3 показано, как это происходит.

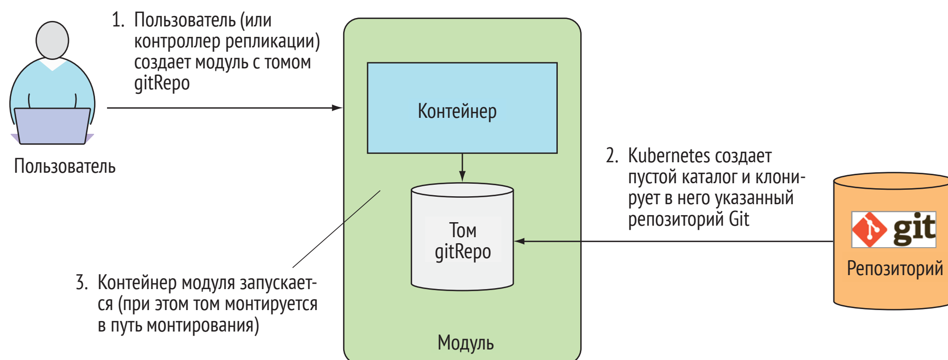


Рис. 6.3. Том gitRepo – это том emptyDir, первоначально заполненный содержимым репозитория Git

ПРИМЕЧАНИЕ. После создания тома `gitRepo` он не синхронизируется с репозиторием, на который ссылается. Файлы в томе при отправке дополнительных фиксаций в репозиторий Git обновляется не будут. Однако если ваш модуль управляется контроллером репликации, удаление модуля приведет к созданию нового модуля, и том этого нового модуля будет содержать последние фиксации.

Например, вы можете использовать репозиторий Git для хранения статических HTML-файлов веб-сайта и создать модуль, содержащий контейнер веб-сервера и том `gitRepo`. Всякий раз, когда модуль создается, он извлекает последнюю версию вашего сайта и начинает его раздавать. Единственным недостатком этого является то, что вам нужно удалять модуль всякий раз, когда вы отправляете изменения в `gitRepo` и хотите начать обслуживать новую версию веб-сайта.

Давай сделаем это прямо сейчас. Это не так уж отличается от того, что вы делали раньше.

Запуск модуля веб-сервера, раздающего файлы из клонированного репозитория Git

Перед созданием модуля вам понадобится репозиторий Git с HTML-файлами. Я создал репо в GitHub по адресу https://github.com/luksa/kubia-website-example.git. Вам нужно будет сделать его форк (создать свою собственную копию репозитория на GitHub), чтобы вы могли позже вносить в него изменения.

После создания форка можно перейти к созданию модуля. На этот раз вам понадобится только один контейнер Nginx и один том `gitRepo` в модуле (не забудьте указать том `gitRepo` на свою вилку моего репозитория), как показано в следующем ниже листинге.

Листинг 6.2. Модуль, использующий том gitRepo: gitrepo-volume-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    gitRepo:                  <- Вы создаете том gitRepo
      repository: https://github.com/luksa/kubia-website-example.git   <- Этот том склонирует репозиторий Git
      revision: master        <- Будет извлечена master-ветка
      directory: .            <- Вы хотите, чтобы репо был клонирован в корневой каталог тома
```

При создании модуля том сначала инициализируется как пустой каталог, а затем в него клонируется указанный репозиторий Git. Если бы вы не задали каталог как `.` (точка), то репозиторий был бы клонирован в подкаталог `kubia-website-example`, что совсем не то, что вы хотите. Вам нужно клонировать репозиторий в корневой каталог вашего тома. Наряду с репозиторием вы также указали, чтобы Kubernetes извлек любую версию, на которую во время создания тома указывает главная ветвь.

При работающем модуле вы можете попытаться попасть в него через перенаправление портов, службу либо путем выполнения команды `curl` изнутри модуля (или любого другого модуля в кластере).

Подтверждение того, что файлы с репозиторием Git не синхронизируются

Теперь вы внесете изменения в файл index.html в вашем репозитории GitHub. Если вы не используете Git локально, то можете отредактировать файл в GitHub напрямую – нажмите на файл в репозитории GitHub, чтобы его открыть, а затем нажмите на значок карандаша, чтобы начать его редактирование. Измените текст, а потом зафиксируйте (commit) изменения, нажав кнопку внизу.

Главная ветвь репозитория Git теперь включает изменения, внесенные в файл HTML. Эти изменения еще не будут видны в веб-сервере Nginx, так как том `gitRepo` не синхронизирован с репозиторием Git. Вы можете подтвердить это, попав в модуль еще раз.

Чтобы увидеть новую версию веб-сайта, необходимо удалить модуль и создать его снова. Вместо того чтобы удалять модуль каждый раз, когда вы вносите изменения, вы можете запустить дополнительный процесс, который держит ваш том в синхронизации с репозиторием Git. Я не буду подробно объяснять, как это сделать. Вместо этого попробуйте сделать это самостоятельно в качестве упражнения, но вот несколько моментов.

Знакомство с побочными контейнерами

Процесс синхронизации с Git должен выполняться не в том же контейнере, что и веб-сервер Nginx, а во втором контейнере: *побочном*. Побочный контейнер представляет собой контейнер, который дополняет работу основного контейнера модуля. Вы добавляете побочный контейнер в модуль, чтобы использовать существующий образ контейнера, вместо того чтобы вводить дополнительную логику в основной код приложения, что делает его слишком сложным и менее пригодным для многоразового использования.

Чтобы найти существующий образ контейнера, который синхронизирует локальный каталог с репозиторием Git, зайдите на Docker Hub и выполните поиск по «git sync». Вы найдете много образов, которые это делают. Затем используйте образ в новом контейнере в модуле из предыдущего примера, смонтируйте существующий том `gitRepo` этого модуля в новый контейнер и сконфигурируйте контейнер синхронизации с Git, чтобы поддерживать фай-

лы в сихронизации с вашим репо Git. Если вы все настроили правильно, то увидите, что файлы, которые раздает веб-сервер, синхронизируются с вашим репозиторием GitHub.

> **ПРИМЕЧАНИЕ.** Пример в главе 18 включает в себя использование контейнера синхронизации с Git, как описано здесь, так что вы можете подождать, пока не достигнете главы 18 и не выполните пошаговые инструкции там, вместо того чтобы делать это упражнение самостоятельно прямо сейчас.

Использование тома gitRepo с приватными репозиториями Git

Есть еще одна причина, для того чтобы прибегать к побочным контейнерам синхронизации с Git. Мы не говорили о том, можно ли использовать том `gitRepo` с приватным репозиторием Git. Оказывается, что нет. Текущий консенсус среди разработчиков Kubernetes заключается в том, чтобы оставить том `gitRepo` простым и не добавлять никакой поддержки для клонирования приватных репозиториев через протокол SSH, потому что это потребует добавления в том `gitRepo` дополнительных параметров конфигурации.

Если вы хотите клонировать приватный репозиторий Git в свой контейнер, вместо тома `gitRepo` вам следует использовать побочный контейнер синхронизации с Git или аналогичный метод.

Краткий вывод о томе gitRepo

Том `gitRepo`, как и том `emptyDir`, – это, в сущности, выделенный каталог, создаваемый специально для модуля и используемый исключительно модулем, который содержит этот том. При удалении модуля том и его содержимое удаляются. Другие типы томов, однако, не создают новый каталог, а вместо этого монтируют существующий внешний каталог в файловую систему контейнера модуля. Содержимое такого тома может выдержать несколько инстанцирований модуля. Далее мы познакомимся с этими типами томов.

6.3 Доступ к файлам в файловой системе рабочего узла

Многие модули должны находиться в неведении о своем узле, поэтому они не должны получать доступ к файлам в файловой системе узла. Но некоторые системные модули (напомним, что они обычно управляются ресурсом DaemonSet) должны либо читать файлы узла, либо использовать файловую систему узла для доступа к устройствам узла через эту файловую систему. Kubernetes делает это возможным посредством тома `hostPath`.

6.3.1 Знакомство с томом hostPath

Том `hostPath` указывает на определенный файл или каталог в файловой системе узла (см. рис. 6.4). Модули, работающие на одном узле и использующие один и тот же путь в томе `hostPath`, видят одни и те же файлы.

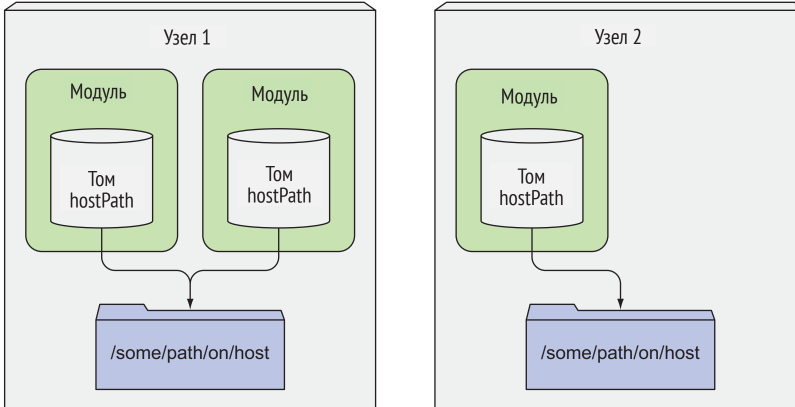


Рис. 6.4. Том hostPath монтирует файл или каталог на рабочем узле в файловую систему контейнера

Тома `hostPath` – это первый тип постоянного хранения, который мы представляем, потому что содержимое томов `gitRepo` и `emptyDir` удаляется, когда модуль демонтируется, в то время как содержимое тома `hostPath` остается. Если модуль удаляется и следующий модуль использует том `hostPath`, который указывает на тот же самый путь на хосте, новый модуль увидит все, что было оставлено предыдущим модулем, но только если он назначен на тот же узел, что и первый модуль.

Если вы думаете, что том `hostPath` можно использовать в качестве места для хранения каталога базы данных, хорошо обдумайте. Поскольку содержимое тома хранится в файловой системе конкретного узла, при переназначении модуля базы данных на другой узел он больше не увидит данных. Это объясняет, почему не рекомендуется использовать том `hostPath` для обычных модулей, поскольку это делает модуль чувствительным к тому, к какому узлу он назначен.

6.3.2 Исследование системных модулей с томами hostPath

Давайте рассмотрим, как правильно использовать том `hostPath`. Вместо того чтобы создавать новый модуль, давайте взглянем, используют ли уже этот тип тома какие-либо существующие общесистемные модули. Как вы помните из предыдущих глав, несколько таких модулей работает в пространстве имен `kube-system`. Давайте выведем их список еще раз:

```
$ kubectl get pod s --namespace kube-system
NAME                          READY   STATUS    RESTARTS   AGE
fluentd-kubia-4ebc2f1e-9a3e   1/1     Running   1          4d
fluentd-kubia-4ebc2f1e-e2vz   1/1     Running   1          31d
```

Выберите первый и посмотрите, какие тома в нем используются (показано в следующем ниже листинге).

Листинг 6.3. Модуль, использующий тома hostPath для доступа к журналам узла

```
$ kubectl describe po fluentd-kubia-4ebc2f1e-9a3e --namespace kube-system
Name:       fluentd-cloud-logging-gke-kubia-default-pool-4ebc2f1e-9a3e
Namespace:  kube-system
...
Volumes:
  varlog:
    Type:   HostPath (bare host directory volume)
    Path:   /var/log
  varlibdockercontainers:
    Type:   HostPath (bare host directory volume)
    Path:   /var/lib/docker/containers
```

> **СОВЕТ.** Если вы используете Minikube, попробуйте модуль `kube-addon-manager-minikube`.

Ага! В модуле используется два тома `hostPath` для доступа к каталогам `/var/log` и `/var/lib/docker/containers` узла. Вы можете подумать, что вам очень повезло с первой попытки найти модуль, использующий том, но не совсем (по крайней мере, не на GKE). Проверьте другие модули, и вы увидите, что для доступа к файлам журналов узла kubeconfig (файлу конфигурации Kubernetes) или сертификатам CA большинство использует этот тип тома.

При проверке других модулей вы увидите, что ни один из них не использует том `hostPath` для хранения собственных данных. Все они используют его, чтобы получать доступ к данным узла. Но, как мы увидим далее в этой главе, тома `hostPath` часто используются для опробирования постоянного хранилища в одноузловых кластерах, например в кластере Minikube. Читайте далее, чтобы узнать о типах томов, которые следует использовать для правильного постоянного хранения данных даже в многоузловом кластере.

> **СОВЕТ.** Помните, что использовать тома `hostPath` следует только в том случае, если вам нужно прочитать или записать системные файлы на узле. Никогда не используйте их для сохранения данных по всем модулям.

6.4 Использование постоянного хранилища

Если приложению, запущенному в модуле, необходимо сохранить данные на диске и иметь те же данные, даже если модуль переназначен на другой узел, вы не можете использовать ни один из типов томов, о которых мы говорили до сих пор. Поскольку эти данные должны быть доступны из любого узла кластера, они должны храниться в сетевом хранилище определенного типа.

Чтобы познакомиться с томами, которые позволяют сохранять данные, вы создадите модуль, который будет работать с документоориентированной ба-

зой данных NoSQL MongoDB. Запуск модуля с базой данных без тома или с несохраняющим томом не имеет смысла, за исключением тестирования, поэтому вы добавите в модуль соответствующий тип тома и смонтируете его в контейнере MongoDB.

6.4.1 Использование постоянного диска GCE Persistent Disk в томе модуля

Если вы выполняли эти примеры в Google Kubernetes Engine, который запускает узлы кластера в Google Compute Engine (GCE), вы будете использовать постоянный диск GCE Persistent Disk в качестве базового механизма хранения.

В ранних версиях система Kubernetes не создавала такого облачного хранилища автоматически – это приходилось делать вручную. Автоматическое создание теперь стало возможным, и вы узнаете об этом позже в данной главе, но сначала вы начнете с создания хранилища вручную. Это даст вам возможность узнать, что именно под этим скрывается.

Создание постоянного диска GCE

Сначала вы создадите постоянный диск GCE. Его необходимо создать в той же зоне, что и кластер Kubernetes. Если вы не помните, в какой зоне вы создали кластер, то можете увидеть ее, выведя список своих кластеров Kubernetes с помощью команды `gcloud`. Это делается следующим образом:

```
$ gcloud container clusters list
NAME    ZONE            MASTER_VERSION   MASTER_IP        ...
kubia   europe-west1-b  1.2.5            104.155.84.137   ...
```

Данный результат показывает, что вы создали свой кластер в зоне `europe-west1-b`, поэтому необходимо создать постоянный диск GCE в той же самой зоне. Диск создается следующим образом:

```
$ gcloud compute disks create --size=1GiB --zone=europe-west1-b mongodb
WARNING: You have selected a disk size of under [200GB]. This may result in
     poor I/O performance. For more information, see:
     https://developers.google.com/compute/docs/disks#pdperformance.
Created [https://www.googleapis.com/compute/v1/projects/rapid-pivot-
     136513/zones/europe-west1-b/disks/mongodb].
NAME      ZONE            SIZE_GB   TYPE          STATUS
mongodb   europe-west1-b  1         pd-standard   READY
```

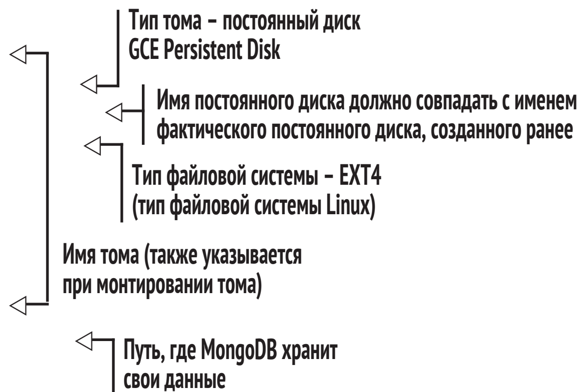
Эта команда создает 1-гигабайтный большой постоянный диск GCE под названием `mongodb`. Вы можете проигнорировать предупреждение о размере диска, потому что вас не волнует производительность диска для тестов, которые вы собираетесь выполнить.

Создание модуля с использованием тома gcePersistentDisk

Теперь, когда физическое хранилище настроено правильно, его можно использовать в томе внутри модуля MongoDB. Для этого модуля вы подготовите YAML, который показан в следующем ниже листинге.

Листинг 6.4. Модуль с использованием тома gcePersistentDisk: mongodb-pod-gcepd.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  volumes:
  - name: mongodb-data
    gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
```



> **ПРИМЕЧАНИЕ.** Если вы используете Minikube, то вы не можете использовать постоянный диск GCE Persistent Disk, но вы можете развернуть `mongodb-pod-hostpath.yaml`, в котором вместо постоянного диска GCE PD используется том `hostPath`.

Модуль содержит единственный контейнер и единственный том, подкрепленный постоянным диском GCE Persistent Disk, который вы создали (как показано на рис. 6.5). Вы монтируете том внутри контейнера в `/data/db`, потому что именно там MongoDB хранит свои данные.

Запись данных в постоянное хранилище путем добавления документов в базу данных MongoDB

После создания модуля и запуска контейнера можно запустить оболочку MongoDB внутри контейнера и использовать ее для записи некоторых данных в хранилище данных. Вы запустите оболочку, как показано в следующем ниже листинге.

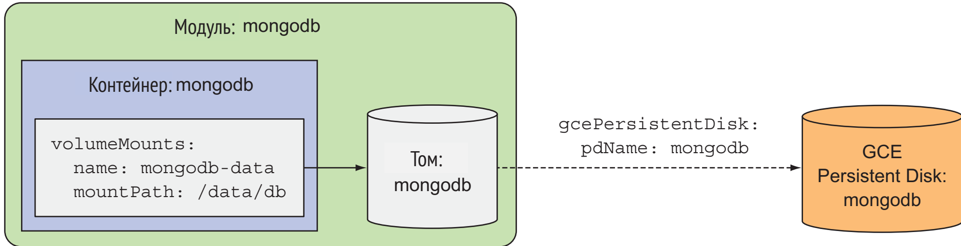


Рис. 6.5. Модуль с единственным контейнером, управляющим MongoDB, который монтирует том, ссылающийся на внешний постоянный диск GCE Persistent Disk

Листинг 6.5. Вход в оболочку MongoDB внутри модуля mongodb

```
$ kubectl exec -it mongodb mongo
MongoDB shell version: 3.2.8
connecting to: mongodb://127.0.0.1:27017
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    http://docs.mongodb.org/
Questions? Try the support group
    http://groups.google.com/group/mongodb-user
...
>
```

MongoDB позволяет хранить документы JSON, поэтому вы сохраните один такой документ, чтобы убедиться, что он сохранился постоянно и может быть получен после того, как модуль будет воссоздан. Вставьте новый документ JSON с помощью следующих ниже команд:

```
> use mystore
switched to db mystore
> db.foo.insert({name:'foo'})
WriteResult({ "nInserted" : 1 })
```

Вы вставили простой документ JSON с единственным свойством (`name: 'foo'`). Теперь примените команду `find()` для просмотра вставленного документа:

```
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cfd512374cc75"), "name" : "foo" }
```

Готово. Теперь документ должен храниться на постоянном диске GCE.

Воссоздание модуля и проверка возможности чтения данных, сохраненных предыдущим модулем

Теперь можно выйти из оболочки `mongodb` (наберите `exit` и нажмите **Enter**), а затем удалить модуль и его воссоздать:

```
$ kubectl delete pod mongodb
pod "mongodb" deleted
$ kubectl create -f mongodb-pod-gcepd.yaml
pod "mongodb" created
```

Новый модуль использует тот же постоянный диск GCE, что и предыдущий модуль, поэтому контейнер MongoDB, запущенный внутри него, должен видеть те же данные, даже если модуль запланирован на другой узел.

> **СОВЕТ.** На какой узел запланирован модуль, можно увидеть, выполнив команду `kubectl get po -o wide`.

После того как контейнер запущен, вы можете снова запустить оболочку MongoDB и проверить, может ли сохраненный ранее документ быть получен, как показано в следующем ниже листинге.

Листинг 6.6. Извлечение сохраненных данных MongoDB в новом модуле

```
$ kubectl exec -it mongodb mongo
MongoDB shell version: 3.2.8
connecting to: mongodb://127.0.0.1:27017
Welcome to the MongoDB shell.
...
> use mystore
switched to db mystore
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cfd512374cc75"), "name" : "foo" }
```

Как и ожидалось, данные по-прежнему там, даже если вы удалили модуль и повторно его создали. Это подтверждает, что вы можете использовать постоянный диск GCE для хранения данных в нескольких экземплярах модуля.

Закончим играть с модулем MongoDB, поэтому можете спокойно его снова удалить, но не удаляйте базовый постоянный диск GCE. Позже в главе вы воспользуетесь им снова.

6.4.2 Использование томов других типов с базовым постоянным хранилищем

Причина, почему вы создали том GCE Persistent Disk, вызвана тем, что кластер Kubernetes работает на движке Google Kubernetes Engine. При запуске кластера в другом месте следует использовать тома других типов в зависимости от базовой инфраструктуры.

Например, если ваш кластер Kubernetes работает на Amazon AWS EC2, то для обеспечения постоянного хранения ваших модулей вы можете использовать том `awsElasticBlockStore`. Если кластер работает в Microsoft Azure, то можно использовать том `azureFile` либо `azureDisk`. Мы не будем вдаваться в подробности о том, как это сделать, но это практически то же самое, что и в предыду-

щем примере. Сначала необходимо создать фактическое базовое хранилище, а затем задать соответствующие свойства в определении тома.

Использование тома постоянного блочного хранилища для AWS

Например, чтобы использовать постоянное блочное хранилище для AWS вместо постоянного диска GCE Persistent Disk, достаточно изменить определение тома, как показано в следующем ниже листинге (см. строки, выделенные жирным шрифтом).

Листинг 6.7. Модуль с использованием тома awsElasticBlockStore: mongodb-pod-aws.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  volumes:
  - name: mongodb-data
    awsElasticBlockStore:      <-- Использование awsElasticBlockStore вместо gcePersistentDisk
    volumeId: my-volume        <-- Указать ИД создаваемого вами тома EBS
    fsType: ext4               <-- Тип файловой системы – EXT4, как и раньше
containers:
- ...
```

Использование тома NFS

Если ваш кластер работает на вашем собственном наборе серверов, у вас есть множество других поддерживаемых вариантов подключения внешнего хранилища внутри вашего тома. Например, для монтирования простого общего ресурса NFS необходимо указать только сервер NFS и путь, экспортируемый сервером, как показано в следующем ниже листинге.

Листинг 6.8. Модуль с использованием тома NFS: mongodb-pod-nfs.yaml

```
volumes:
- name: mongodb-data
  nfs:                     <-- Этот том поддерживается общим ресурсом NFS
    server: 1.2.3.4        <-- IP-адрес сервера NFS
    path: /some/path       <-- Путь, экспортируемый сервером
```

Использование других технологий хранения

Другие поддерживаемые варианты включают дисковый ресурс ISCSI, `glusterfs` для точки монтирования GlusterFS, `rbd` для блочного устройства RADOS Block Device, `flexVolume`, `cinder`, `cephfs`, `flocker`, `fc` (оптоволоконный ка-

нал Fibre Channel) и др. Вам не нужно знать их все, если вы их не используете. Они упоминаются здесь, чтобы показать вам, что Kubernetes поддерживает широкий спектр технологий хранения и вы можете использовать то, что вы предпочитаете и к чему привыкли.

Для того чтобы просмотреть сведения о том, какие свойства необходимо задавать для каждого из этих типов томов, можно обратиться к определениям API Kubernetes в справочнике по API Kubernetes или просмотреть сведения с помощью команды `kubectl explain`, как показано в главе 3. Если вы уже знакомы с определенной технологией хранения, использование команды `explain` должно позволить вам легко понять, как смонтировать том соответствующего типа и использовать его в ваших модулях.

Но должен ли разработчик знать все это? Должен ли разработчик при создании модуля иметь дело с деталями хранения, связанными с инфраструктурой, или же все это должно быть оставлено администратору кластера?

Наличие тех или иных томов модуля относится к фактической базовой инфраструктуре, это совсем не то, чем занимается Kubernetes, не так ли? Например, вменять в обязанности разработчика указывать имя сервера NFS было бы неправильно. И это еще не самое страшное.

Включение этого типа информации, связанной с инфраструктурой, в определение модуля означает, что определение модуля в значительной степени привязывается к конкретному кластеру Kubernetes. Вы не можете использовать то же самое определение модуля в другом кластере. Вот почему использование таких томов не является наилучшим способом привязки постоянного хранилища к модулям. Вы узнаете, как улучшить эту ситуацию, в следующем далее разделе.

6.5 Отделение модулей от базовой технологии хранения

Все типы постоянных томов, которые мы исследовали до сих пор, требовали от разработчика модуля знаний о фактической инфраструктуре сетевого хранилища, имеющейся в кластере. Например, для того чтобы создать том с поддержкой NFS, разработчик должен знать фактический сервер, на котором расположен экспорт NFS. Это противоречит основной идее Kubernetes, которая направлена на то, чтобы скрыть фактическую инфраструктуру как от приложения, так и от его разработчика, освобождая их от необходимости беспокоиться о специфике инфраструктуры и делая приложения переносимыми по широкому спектру облачных провайдеров и локальных центров обработки данных.

В идеале, разработчик, развертывающий свои приложения в Kubernetes, никогда не должен знать, какая технология хранения используется в основе, так же, как он не должен знать, какой тип физических серверов используется для запуска его модулей. Операции, связанные с инфраструктурой, должны быть исключительно предметом ведения администратора кластера.

Когда разработчику требуется определенный объем постоянного хранилища для своего приложения, он может подать заявку на него в Kubernetes так же, как он может запросить ЦП, память и другие ресурсы при создании модуля. Системный администратор может настроить кластер так, чтобы он мог предоставлять приложениям то, что они запрашивают.

6.5.1 Знакомство с томами PersistentVolume и заявками PersistentVolumeClaim

Для того чтобы позволить приложениям запрашивать хранилище в кластере Kubernetes без необходимости иметь дело со спецификой инфраструктуры, были введены два новых ресурса. Это постоянный том (PersistentVolume) и заявка на постоянный том (PersistentVolumeClaim). Эти имена могут вводить в заблуждение, поскольку, как вы видели в предыдущих томах, даже обычные тома Kubernetes могут использоваться для постоянного хранения данных.

Использовать постоянный том PersistentVolume внутри модуля немного сложнее, чем использовать обычный том модуля, поэтому давайте на рис. 6.6 проиллюстрируем, каким образом связаны между собой модули, заявки на постоянный том PersistentVolumeClaim, постоянные тома PersistentVolume и фактическое базовое хранилище.

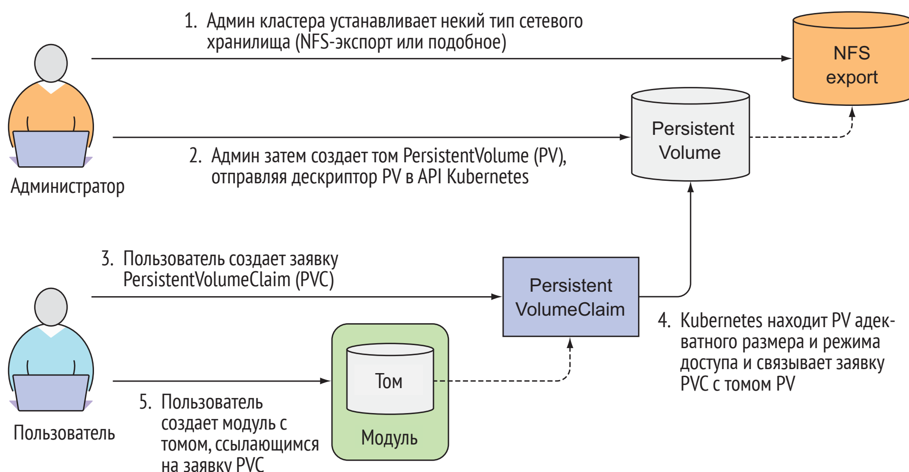


Рис. 6.6. Постоянные тома PersistentVolume резервируются администраторами кластера и потребляются модулями посредством заявок PersistentVolumeClaim

Вместо того чтобы разработчик добавлял в свой модуль том, относящийся к конкретной технологии, администратор кластера настраивает базовое хранилище, а затем регистрирует его в Kubernetes, создавая ресурс PersistentVolume через сервер API Kubernetes. При создании ресурса PersistentVolume админ определяет его размер и режимы доступа.

Когда пользователю кластера необходимо использовать постоянное хранилище в одном из своих модулей, он сначала создает манифест с заявкой Persis-

tentVolumeClaim, указывая минимальный размер и требуемый режим доступа. Затем пользователь отправляет манифест с заявкой PersistentVolumeClaim в API-сервер Kubernetes, и Kubernetes находит соответствующий ресурс PersistentVolume и связывает его с заявкой.

Заявка PersistentVolumeClaim затем может использоваться как один из томов в модуле. Другие пользователи не могут использовать тот же том PersistentVolume до тех пор, пока он не будет высвобожден путем удаления связанной заявки PersistentVolumeClaim.

6.5.2 Создание ресурса PersistentVolume

Давайте вернемся к примеру с MongoDB, но, в отличие от предыдущего примера, вы не будете ссылаться на постоянный диск GCE Persistent Disk в модуле напрямую. Вместо этого вы сначала возьмете на себя роль администратора кластера и создадите ресурс PersistentVolume, поддерживаемый постоянным диском GCE Persistent Disk. Затем вы возьмете на себя роль разработчика приложения и сначала заявите PersistentVolume, а затем используете его внутри своего модуля.

В разделе 6.4.1 вы настраивали физическое хранилище путем создания постоянного диска GCE Persistent Disk, поэтому вам не нужно делать это снова. Все, что вам нужно сделать, – это создать ресурс PersistentVolume в Kubernetes, подготовив манифест, показанный в следующем ниже листинге, и разместить его на сервере API.

Листинг 6.9. Постоянный том PersistentVolume gcePersistentDisk: mongodb-pv-gcepd.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

Определение размера PersistentVolume (capacity)

Он может быть смонтирован либо одиночным клиентом для чтения и записи, либо многочисленными клиентами только для чтения (accessModes)

После высвобождения заявки PersistentVolume должен быть сохранен (не стерт и не удален) (persistentVolumeReclaimPolicy)

PersistentVolume поддерживается постоянным диском GCE Persistent Disk, созданным ранее (gcePersistentDisk)

ПРИМЕЧАНИЕ. Если вы применяете Minikube, создайте ресурс PV, используя файл `mongodb-pv-hostpath.yaml`.

При создании ресурса PersistentVolume администратор должен сообщить Kubernetes, какова его емкость и предназначен ли он для чтения и/или записи единственным узлом или несколькими узлами одновременно. Он также должен сообщить Kubernetes, что делать с ресурсом PersistentVolume, когда он высвобождается (когда заявка PersistentVolumeClaim, к которой он привязан, удаляется). И последнее, но, разумеется, не менее важное, он должен указать тип, расположение и другие свойства фактического хранения, которые поддерживает этот ресурс PersistentVolume. Если вы посмотрите внимательно, эта последняя часть точно такая же, как и ранее, когда вы напрямую ссылались на GCE Persistent Disk в томе модуля (снова показано в следующем ниже листинге).

Листинг 6.10. Ссылка на GCE PD в томе модуля

```
spec:
  volumes:
  - name: mongodb-data
    gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
...
```

После создания тома PersistentVolume с помощью команды `kubectl create` он должен быть готов к размещению заявки. Убедитесь, что это так, выведя список всех томов PersistentVolume:

```
$ kubectl get pv
NAME         CAPACITY   RECLAIMPOLICY   ACCESSMODES   STATUS      CLAIM
mongodb-pv   1Gi        Retain          RWO,ROX       Available
```

> **ПРИМЕЧАНИЕ.** Несколько столбцов опущено. Кроме того, символы `pv` используются как аббревиатура для `persistentvolume`.

Как и ожидалось, PersistentVolume отображается как доступный (`Available`), поскольку вы еще не создали заявку PersistentVolumeClaim.

> **ПРИМЕЧАНИЕ.** Ресурсы PersistentVolume не принадлежат ни к какому пространству имен (см. рис. 6.7). Это ресурсы кластерного уровня, такие как узлы.

6.5.3 Подача заявки на PersistentVolume путем создания ресурса PersistentVolumeClaim

Теперь давайте сложим наши полномочия админа и снова возложим на себя обязанности разработчика. Предположим, необходимо развернуть мо-

дуль, требующий постоянного хранилища. Вы будете использовать постоянный том PersistentVolume, созданный ранее. Но вы не можете использовать его непосредственно в модуле. Сначала вы должны заявить об этом.

Подача заявки на PersistentVolume – это совершенно другой процесс, отдельный от создания модуля, потому что вы хотите, чтобы та же самая заявка PersistentVolumeClaim оставалась в наличии, даже если модуль будет переназначен (напомним, переназначение означает, что предыдущий модуль удаляется и создается новый).

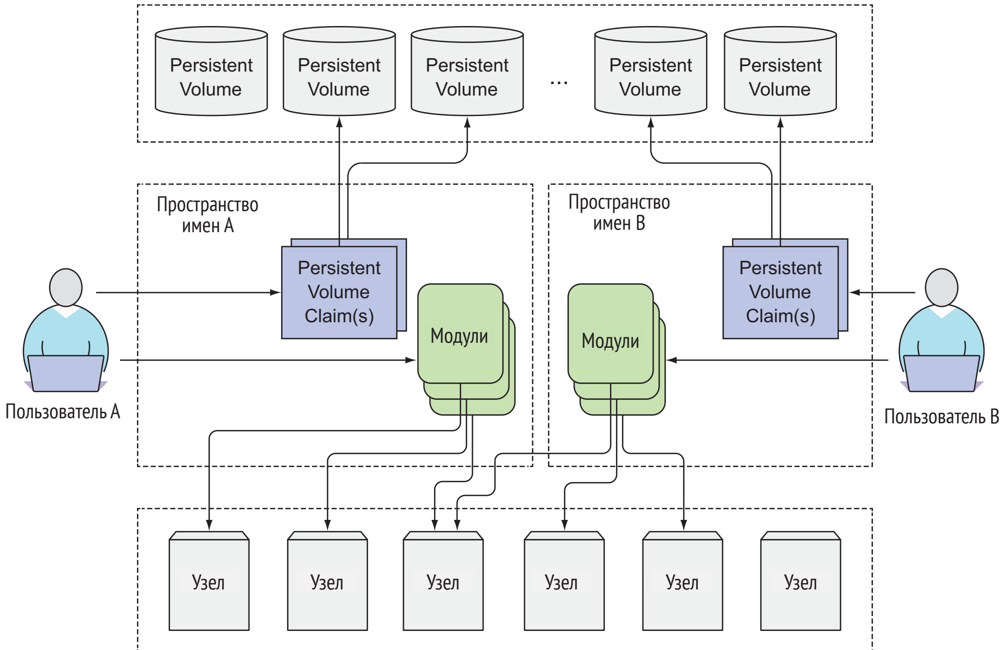


Рис. 6.7. Ресурсы PersistentVolume, такие как узлы кластера, не принадлежат ни к одному пространству имен, в отличие от модулей и ресурсов PersistentVolumeClaim

Создание заявки PersistentVolumeClaim

Теперь вы создадите заявку. Необходимо подготовить манифест PersistentVolumeClaim, как показано в следующем ниже листинге, и отправить его в API Kubernetes с помощью команды `kubectl create`.

Листинг 6.11. Заявка PersistentVolumeClaim: mongodb-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  resources:
```

Имя вашей заявки – оно вам понадобится позже при использовании заявки в качестве тома модуля

```
    requests:
      storage: 1Gi        <-- Запрос хранилища объемом 1 Гб
  accessModes:
  - ReadWriteOnce         <-- Вы хотите, чтобы хранилище поддерживало одного
                              клиента (выполняло операции чтения и записи)
  storageClassName: ""    <-- Вы узнаете об этом в томе
                              о динамическом резервировании
```

Как только вы создаете заявку, Kubernetes находит соответствующий PersistentVolume и связывает его с заявкой. Емкость ресурса PersistentVolume должна быть достаточно большой, чтобы удовлетворить запросы в заявке. Кроме того, режимы доступа тома должны включать режимы доступа, запрошенные в заявке. В вашем случае в заявке запрашивается 1 Гб хранилища и режим доступа ReadWriteOnce. Ресурс PersistentVolume, созданный ранее, соответствует этим двум требованиям, поэтому он привязывается к вашей заявке. Это можно увидеть, проинспектировав заявку.

Вывод списка заявок PersistentVolumeClaim

Выведите список всех заявок PersistentVolumeClaim, чтобы увидеть состояние вашей заявки PVC:

```
$ kubectl get pvc
NAME          STATUS   VOLUME       CAPACITY   ACCESSMODES   AGE
mongodb-pvc   Bound    mongodb-pv   1Gi        RWO,ROX       3s
```

> **ПРИМЕЧАНИЕ.** Символы `pvc` используются в качестве аббревиатуры для `persistentvolumeclaim`.

Заявка показана как связанная (`Bound`) с ресурсом PersistentVolume `mongodb-pv`. Обратите внимание на аббревиатуры, используемые для режимов доступа:

- `RWO` – `ReadWriteOnce` – только один узел может монтировать том для чтения и записи;
- `ROX` – `ReadOnlyMany` – несколько узлов могут монтировать том для чтения;
- `RWX` – `ReadWriteMany` – несколько узлов могут монтировать том как для чтения, так и для записи.

> **ПРИМЕЧАНИЕ.** `RWO`, `ROX` и `RWX` относятся к числу рабочих узлов, которые могут использовать том одновременно, а не к числу модулей!

Вывод списка томов PersistentVolume

Вы также можете увидеть, что том PersistentVolume теперь связан (`Bound`) и его больше нет в наличии (`Available`), проверив его с помощью команды `kubectl get`:

```
$ kubectl get pv
NAME        CAPACITY  ACCESSMODES  STATUS  CLAIM                AGE
mongodb-pv  1Gi       RWO,ROX      Bound   default/mongodb-pvc  1m
```

Том PersistentVolume показывает, что он связан с заявкой `default/mongodb-pvc`. Префикс `default` является пространством имен, в котором расположена заявка (вы создали заявку в пространстве по умолчанию). Мы уже говорили, что ресурсы PersistentVolume находятся в области кластера и поэтому не могут быть созданы в каком-то определенном пространстве имен, однако ресурсы PersistentVolumeClaim могут быть созданы только в конкретном пространстве имен. И затем они могут использоваться лишь модулями в том же пространстве имен.

6.5.4 Использование заявки PersistentVolumeClaim в модуле

Том PersistentVolume теперь ваш, и вы можете его использовать. Никто другой не может претендовать на тот же том до тех пор, пока вы его не высвободите. Как показано в следующем ниже листинге, для того чтобы использовать его внутри модуля, необходимо сослаться на заявку PersistentVolumeClaim по имени внутри тома модуля (да, именно PersistentVolumeClaim, а не напрямую PersistentVolume!).

Листинг 6.12. Модуль, использующий том PersistentVolumeClaim: mongodb-pod-pvc.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: mongodb-pvc
```

Ссылка на PersistentVolumeClaim по имени в томе модуля

Продолжайте и создайте модуль. Теперь проверьте, действительно ли модуль использует тот же том PersistentVolume и его базовый GCE PD. Вы должны увидеть сохраненные ранее данные, снова запустив оболочку MongoDB. Это показано в следующем ниже листинге.

Листинг 6.13. Извлечение сохраненных данных MongoDB в модуле с использованием PVC и CPVC

```
$ kubectl exec -it mongodb mongo
MongoDB shell version: 3.2.8
connecting to: mongodb://127.0.0.1:27017
Welcome to the MongoDB shell.
...
> use mystore
switched to db mystore
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cfd512374cc75"), "name" : "foo" }
```

А вот и он. Вы смогли получить документ, который вы ранее сохранили в MongoDB.

6.5.5 Преимущества использования томов PersistentVolume и заявок

Изучите рис. 6.8, показывающий оба способа, которыми модуль может использовать постоянный диск GCE PersistentDisk, – непосредственно либо через PersistentVolume и заявку.

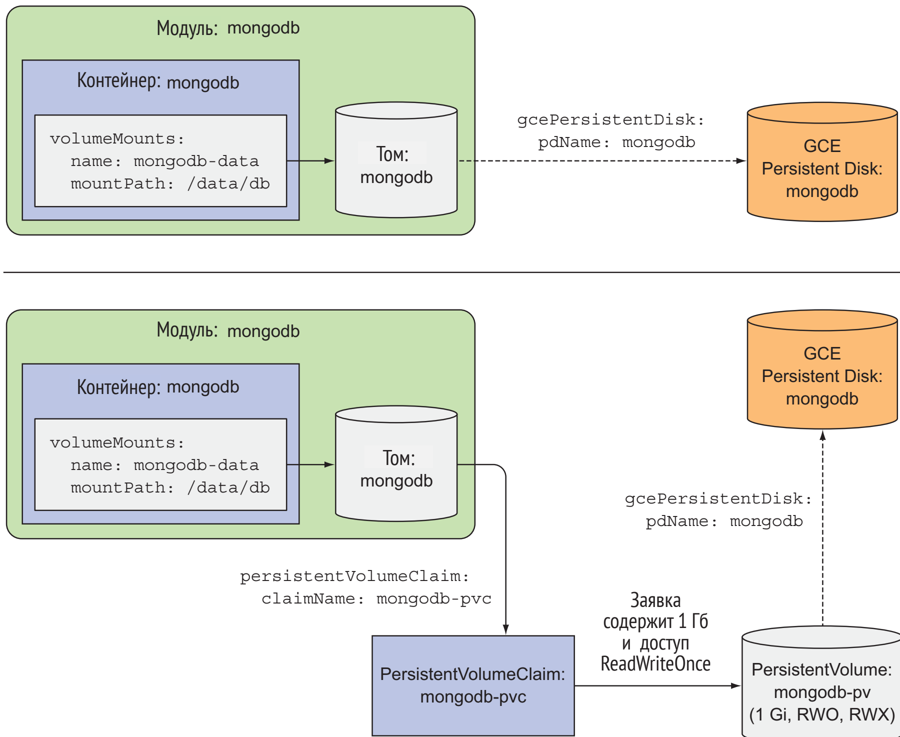


Рис. 6.8. Использование GCE PersistentDisk напрямую либо через PVC и CPVC

Обратите внимание, насколько использование этого косвенного метода получения хранилища из инфраструктуры намного проще для разработчика приложения (или пользователя кластера). Да, это требует дополнительных шагов создания ресурсов PersistentVolume и PersistentVolumeClaim, но разработчику не приходится ничего знать о фактической технологии хранения, используемой внутри.

Кроме того, одни и те же манифесты модуля и заявки теперь могут использоваться в самых разных кластерах Kubernetes, поскольку они не относятся ни к какой конкретной инфраструктуре. В заявке говорится «мне нужен *x* объем хранилища, и я должен иметь возможность, чтобы один клиент мог одновременно читать из него и писать в него», и затем модуль ссылается на заявку по имени в одном из своих томов.

6.5.6 Повторное использование постоянных томов

Прежде чем вы закончите этот раздел о томах PersistentVolume, давайте выполним один последний быстрый эксперимент. Удалите модуль и заявку PersistentVolumeClaim:

```
$ kubectl delete pod mongodb
pod "mongodb" deleted
$ kubectl delete pvc mongodb-pvc
persistentvolumeclaim "mongodb-pvc" deleted
```

А что, если создать заявку PersistentVolumeClaim снова? Будет ли она связана с томом PersistentVolume или нет? Что покажет команда `kubectl get pvc` после создания заявки?

```
$ kubectl get pvc
NAME         STATUS    VOLUME   CAPACITY   ACCESSMODES   AGE
mongodb-pvc  Pending                                     13s
```

Статус заявки показан как находящийся на рассмотрении (`Pending`). Интересно. Когда вы ранее создали заявку, она немедленно была связана с томом PersistentVolume, так почему же не сейчас? Может быть, вывод списка томов PersistentVolume прольет больше света на причину:

```
$ kubectl get pv
NAME        CAPACITY  ACCESSMODES  STATUS    CLAIM REASON         AGE
mongodb-pv  1Gi       RWO,ROX      Released  default/mongodb-pvc  5m
```

Столбец `STATUS` показывает, что PersistentVolume высвобожден (`Released`) и отсутствует (`Available`), как и раньше. Поскольку этот том вы уже использовали, он может содержать данные и не должен быть привязан к совершенно новой заявке без предоставления администратору кластера возможности его очистить. Без этого новый модуль, использующий тот же том PersistentVolume, мог бы прочитать данные, сохраненные там предыдущим модулем, даже если заявка и модуль были созданы в другом пространстве имен (и поэтому, вероятно, принадлежат другому клиенту кластера).

Повторное использование постоянных томов вручную

Вы сообщили Kubernetes, чтобы ваш том PersistentVolume вел себя таким образом во время его создания – присвоив принадлежащей ему политике рекламации `persistentVolumeReclaimPolicy` значение `Retain`. Вы хотели, чтобы Kubernetes удерживал том и его содержимое, после того как он будет высвобожден из своей заявки. Насколько мне известно, единственный способ вручную повторно использовать постоянный том PersistentVolume, чтобы сделать его доступным снова, состоит в том, чтобы удалить и воссоздать ресурс PersistentVolume. Когда вы будете это делать, на ваше усмотрение остается решение, что делать с файлами в базовом хранилище: вы можете их удалить или оставить их в покое, чтобы они могли быть повторно использованы следующим модулем.

Повторное использование постоянных томов автоматически

Существуют две другие возможные политики рекламации: `Recycle` и `Delete`. Первая удаляет содержимое тома и делает его доступным для повторной заявки. Благодаря этому ресурс PersistentVolume может быть повторно использован несколько раз разными заявками PersistentVolumeClaim и разными модулями, как видно по рис. 6.9.

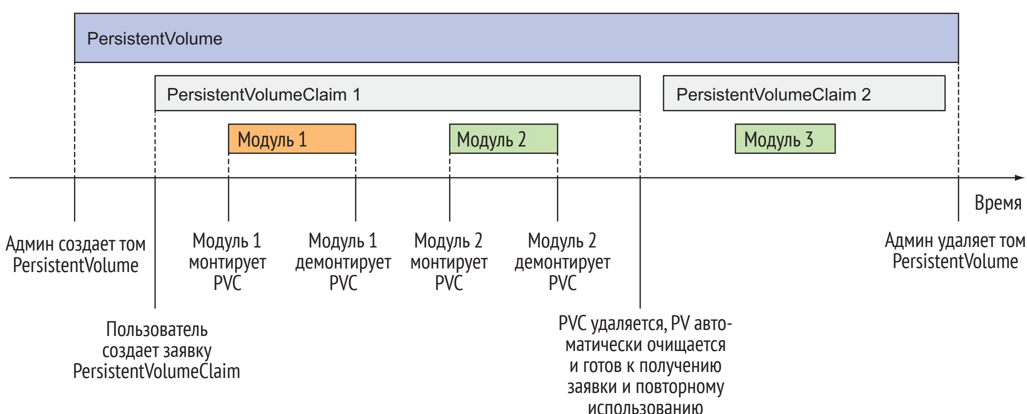


Рис. 6.9. Срок службы тома PersistentVolume, заявок PersistentVolumeClaim и их использующих модулей

С другой стороны, политика `Delete` удаляет базовое хранилище. Обратите внимание, что параметр `Recycle` в настоящее время недоступен для постоянных дисков GCE Persistent Disk. Этот тип тома PersistentVolume поддерживает только политики удержания `Retain` или удаления `Delete`. Другие типы томов PersistentVolume могут поддерживать или не поддерживать каждый из этих параметров, поэтому перед созданием собственного тома PersistentVolume проверьте, какие политики возобновления поддерживаются для конкретного базового хранилища, которое будет использоваться в томе.

СОВЕТ. Политику рекламации тома PersistentVolume можно изменить на существующем томе PersistentVolume. Например, если она изначально настроена на удаление (`Delete`), ее можно легко поменять на удержание (`Retain`), чтобы предотвратить потерю ценных данных.

6.6 Динамическое резервирование томов PersistentVolume

Вы видели, как использование ресурсов PersistentVolume и PersistentVolumeClaim позволяет легко получить постоянное хранилище без необходимости разработчику иметь дело с реальными технологиями хранения, лежащими в основе. Но для этого по-прежнему требуется администратор кластера, который предварительно резервирует фактическое хранилище. К счастью, Kubernetes также может выполнять это задание автоматически с помощью динамического резервирования томов PersistentVolume.

Администратор кластера, вместо того чтобы создавать ресурсы PersistentVolume, может развернуть поставщика (provisioner) ресурса PersistentVolume и определить один или более объектов StorageClass, чтобы позволить пользователям выбрать, какой тип ресурса PersistentVolume они хотят. Пользователи могут ссылаться на класс хранилища StorageClass в своих заявках PersistentVolumeClaim, и поставщик будет принимать это во внимание при резервировании постоянного хранилища.

ПРИМЕЧАНИЕ. Так же, как ресурсы PersistentVolume, ресурсы StorageClass не имеют пространств имен.

Kubernetes включает в себя поставщиков для самых популярных облачных провайдеров, поэтому администратору не всегда нужно создавать поставщиков. Но если Kubernetes развертывается локально, то необходимо создать своего собственного поставщика.

Вместо предварительного резервирования администратор кластера может развернуть поставщика ресурсов PersistentVolume, определить один или два (или больше) ресурсов StorageClass и позволить системе создавать новый ресурс PersistentVolume каждый раз, когда один из них запрашивается посредством заявки PersistentVolumeClaim. Самое замечательное в этом то, что ресурсы PersistentVolume невозможно исчерпать (разумеется, вы можете исчерпать пространство для хранения).

6.6.1 Определение доступных типов хранилища с помощью ресурсов StorageClass

Прежде чем пользователь сможет создать заявку PersistentVolumeClaim, которая приведет к резервированию нового ресурса PersistentVolume, адми-

нистратору необходимо создать один или несколько ресурсов StorageClass. Рассмотрим пример одного из них в следующем ниже листинге.

Листинг 6.14. Определение ресурса StorageClass: storageclass-fast-gcepd.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd    <-- Плагин тома, используемый для резервирования ресурса PersistentVolume
parameters:
  type: pd-ssd                       <-- Параметры, передаваемые поставщику
  zone: europe-west1-b
```

ПРИМЕЧАНИЕ. При использовании Minikube следует задействовать файл storageclass-fast-hostpath.yaml.

Ресурс StorageClass указывает, какой поставщик (provisioner) должен использоваться для создания ресурса PersistentVolume, когда заявка PersistentVolumeClaim запрашивает этот ресурс StorageClass. Параметры, определенные в определении StorageClass, передаются поставщику и специфичны для каждого плагина поставщика.

6.6.2 Запрос на класс хранилища в заявке PersistentVolumeClaim

После создания ресурса StorageClass пользователи могут в своих заявках PersistentVolumeClaim ссылаться на класс хранилища по имени.

Создание заявки PersistentVolumeClaim с запросом на определенный класс хранения

Вы можете изменить свою заявку mongodb-pvc для использования динамического резервирования. Следующий ниже листинг показывает обновленное определение ресурса заявки PVC в YAML.

Листинг 6.15. Определение заявки PVC с динамическим резервированием: mongodb-pvc-dp.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: fast    <-- Эта заявка PVC запрашивает вручную созданный класс хранилища
  resources:
```

```
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

Помимо указания размера и режимов доступа, заявка PersistentVolumeClaim теперь также указывает на класс хранилища, который вы хотите использовать. Когда вы создаете заявку, ресурс PersistentVolume образуется поставщиком, ссылка на который указана в ресурсе StorageClass `fast`. Поставщик используется, даже если существующий зарезервированный вручную ресурс PersistentVolume совпадает с заявкой PersistentVolumeClaim.

> **ПРИМЕЧАНИЕ.** Если в заявке PVC сослаться на несуществующий класс хранения, резервирование ресурса PV завершится ошибкой (при использовании команды `kubectl describe` на заявке PVC вы увидите событие `ProvisioningFailed`).

Исследование созданной заявки PVC и динамически созданного ресурса PV

Далее вы создадите заявку PVC, а затем примените команду `kubectl get`, чтобы это увидеть:

```
$ kubectl get pvc mongodb-pvc
NAME         STATUS  VOLUME        CAPACITY  ACCESSMODES  STORAGECLASS
mongodb-pvc  Bound   pvc-1e6bc048  1Gi       RWO          fast
```

Столбец `VOLUME` показывает ресурс PersistentVolume, привязанный к этой заявке (фактическое имя длиннее, чем показано выше). Вы можете теперь попробовать вывести список ресурсов PersistentVolume, чтобы увидеть, что новый ресурс PV действительно был создан автоматически:

```
$ kubectl get pv
NAME          CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS    STORAGECLASS
mongodb-pv    1Gi       RWO,ROX      Retain         Released
pvc-1e6bc048  1Gi       RWO          Delete         Bound     fast
```

> **ПРИМЕЧАНИЕ.** Показаны только соответствующие столбцы.

Вы можете увидеть динамически зарезервированный ресурс PersistentVolume. Его заданный размер и режимы доступа – это то, что вы заявили в заявке PVC. Его политика восстановления установлена в `Delete`, то есть ресурс PersistentVolume будет удален, когда заявка PVC будет удалена. Помимо ресурса PV, поставщик также зарезервировал фактическое хранилище. Ваш ресурс StorageClass `fast` настроен на использование поставщика `kubernetes.io/gce-pd`, который резервирует постоянные диски GCE Persistent Disk. Вы можете увидеть диск с помощью следующей ниже команды:

```
$ gcloud compute disks list
NAME                        ZONE            SIZE_GB  TYPE         STATUS
gke-kubia-dyn-pvc-1e6bc048  europe-west1-d  1        pd-ssd       READY
gke-kubia-default-pool-71df europe-west1-d  100      pd-standard  READY
gke-kubia-default-pool-79cd europe-west1-d  100      pd-standard  READY
gke-kubia-default-pool-blc4 europe-west1-d  100      pd-standard  READY
mongodb                     europe-west1-d  1        pd-standard  READY
```

Как видно из результата, имя первого постоянного диска предполагает, что он был зарезервирован динамически, и его тип показывает, что это SSD, как указано в классе хранилища, который вы создали ранее.

Как использовать классы хранилищ

Администратор кластера может создать несколько классов хранилищ с различной производительностью или другими характеристиками. Затем разработчик решает, какой из них наиболее подходит для каждой заявки, которую он создает.

Приятная вещь относительно ресурсов StorageClass состоит в том, что заявки ссылаются на них по имени. Пока имена StorageClass остаются одинаковыми во всех из них, определения заявок PVC, следовательно, переносимы на различные кластеры. Чтобы увидеть эту переносимость самому, вы можете попробовать запустить тот же пример на Minikube, если до этого момента вы использовали GKE. Администратору кластера необходимо создать другой класс хранения (но с тем же именем). Класс хранения, указанный в файле `storageclass-fasthostpath.yaml`, специально сделан для использования в Minikube. Затем, как только вы развертываете класс хранения, вы как пользователь кластера можете развернуть тот же самый манифест заявки PVC и тот же самый манифест модуля, как раньше. Это показывает, как модули и заявки PVC переносимы на разные кластеры.

6.6.3 Динамическое резервирование без указания класса хранилища

По мере продвижения по этой главе присоединение постоянного хранилища к модулям становилось все проще. Разделы данной главы отражают эволюцию резервирования хранилища начиная с ранних версий Kubernetes до настоящего времени. В этом последнем разделе мы рассмотрим самый последний и простой способ присоединения ресурса PersistentVolume к модулю.

Вывод списка классов хранения

При создании самостоятельно созданного класса хранилища `fast` вы не проверили, определены ли в кластере какие-либо существующие классы хранилищ. Почему бы не сделать это сейчас? Вот классы хранилищ, доступные в GKE:

```
$ kubectl get sc
NAME                TYPE
fast                kubernetes.io/gce-pd
standard (default)  kubernetes.io/gce-pd
```

> **ПРИМЕЧАНИЕ.** Символы `sc` используются в качестве аббревиатуры для `storageclass`.

Помимо класса хранения `fast`, который вы создали сами, существует стандартный класс хранения `standard`, который помечен как класс, существующий по умолчанию. Вы узнаете, что это значит, через минуту. Давайте выведем список классов хранения в Minikube, чтобы можно было сравнить:

```
$ kubectl get sc
NAME                TYPE
Fast                k8s.io/minikube-hostpath
standard (default)  k8s.io/minikube-hostpath
```

Опять-таки, класс хранения `fast` был создан вами, и класс хранения `standard`, существующий по умолчанию, здесь тоже есть. Сравнивая столбцы `TYPE` в двух списках, вы видите, что GKE использует поставщика `kubernetes.io/gce-pd`, в то время как Minikube применяет `k8s.io/minikube-hostpath`.

Исследование класса хранилища, существующего по умолчанию

Для получения дополнительной информации о стандартном классе хранилища в кластере GKE вы примените команду `kubectl get`. Это показано в следующем ниже листинге.

Листинг 6.16. Определение стандартного класса хранилища в GKE

```
$ kubectl get sc standard -o yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
  creationTimestamp: 2017-05-16T15:24:11Z
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
    kubernetes.io/cluster-service: "true"
  name: standard
  resourceVersion: "180"
  selfLink: /apis/storage.k8s.io/v1/storageclassesstandard
  uid: b6498511-3a4b-11e7-ba2c-42010a840014
parameters:
  type: pd-standard
provisioner: kubernetes.io/gce-pd
```

Эта аннотация помечает класс хранилища как существующий по умолчанию

Параметр type используется поставщиком, чтобы узнать, какой тип GCE Persistent Disk создавать

Поставщик GCE Persistent Disk используется для резервирования ресурсов PV этого класса

Если присмотреться к началу списка, то вы увидите, что определение класса хранения включает аннотацию, которая делает его классом хранилища по умолчанию. Класс хранилища по умолчанию используется для динамического резервирования ресурса PersistentVolume, если заявка PersistentVolumeClaim явно не указывает, какой класс хранения использовать.

Создание заявки PersistentVolumeClaim без указания класса хранилища

Вы можете создать заявку PVC без указания атрибута `storageClassName`, и (на Google Kubernetes Engine) для вас будет зарезервирован постоянный диск GCE Persistent Disk с типом `pd-standard`. Попробуйте это сделать, создав заявку из YAML, как в следующем ниже листинге.

Листинг 6.17. Заявка PVC без определения класса хранилища: mongodb-pvc-dp-nostorageclass.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc2
spec:
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

Вы не указываете атрибут storageClassName (в отличие от предыдущих примеров)

Это определение заявки PVC включает только запрос размера хранилища и желаемые режимы доступа, но не класс хранилища. При создании заявки PVC будет применяться любой класс хранилища, помеченный как используемый по умолчанию. Вы можете подтвердить, что это так:

```
$ kubectl get pvc mongodb-pvc2
NAME          STATUS  VOLUME        CAPACITY  ACCESSMODES  STORAGECLASS
mongodb-pvc2  Bound   pvc-95a5ec12  1Gi       RWO          standard

$ kubectl get pv pvc-95a5ec12
NAME          CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS  STORAGECLASS
pvc-95a5ec12  1Gi       RWO          Delete         Bound   standard

$ gcloud compute disks list
NAME                        ZONE            SIZE_GB  TYPE         STATUS
gke-kubia-dyn-pvc-95a5ec12  europe-west1-d  1        pd-standard  READY
...
```

Принудительная привязка заявки PersistentVolumeClaim к одному из заранее зарезервированных ресурсов PersistentVolume

Это, наконец, приводит нас к вопросу, почему вы присваиваете атрибуту `storageClassName` пустую строку в листинге 6.11 (когда требовалось, чтобы заявка PVC была привязана к ресурсу PV, который вы зарезервировали вручную). Давайте я здесь повторю соответствующие строки этого определения заявки PVC:

```
kind: PersistentVolumeClaim
spec:
  storageClassName: ""
```

Указание пустой строки в качестве имени класса хранилища обеспечивает привязку заявки PVC к предварительно зарезервированному ресурсу PV вместо динамического резервирования нового ресурса

Если бы вы не присвоили атрибуту storageClassName пустую строку, то динамический поставщик томов зарезервировал бы новый ресурс PersistentVolume, несмотря на наличие соответствующего предварительно зарезервированного ресурса PersistentVolume. В том месте я хотел продемонстрировать, как заявка привязывается к предварительно зарезервированному вручную ресурсу PersistentVolume. Я не хотел, чтобы динамический поставщик вмешивался.

> **СОВЕТ.** Если вы хотите, чтобы заявка PVC использовала предварительно зарезервированный ресурс PersistentVolume, тогда явным образом присвойте `storageClassName` значение `""`.

Полная картина динамического резервирования ресурса PersistentVolume

Это подводит нас к концу данной главы. Подводя итог, следует еще раз отметить, что самый лучший способ присоединить постоянное хранилище к модулю – создать PVC (при необходимости с `storageClassName`, заданным явным образом) и модуль (который ссылается на PVC по имени). Обо всем остальном позаботится динамический поставщик ресурса PersistentVolume.

Чтобы получить полную картину шагов, вовлеченных в получение динамически резервируемого PersistentVolume, исследуйте рис. 6.10.

6.7 Резюме

В этой главе было показано, как тома используются для обеспечения контейнеров модуля временным либо постоянным хранилищем. Вы узнали, как:

- создавать многоконтейнерный модуль и делать так, чтобы контейнеры модуля оперировали с одними и теми же файлами путем добавления тома в модуль и его монтирования в каждом контейнере;
- использовать том `emptyDir` для хранения временных данных;
- использовать том `gitRepo` для того, чтобы легко заполнять каталог содержимым репозитория Git при запуске модуля;

- использовать том `hostPath`, чтобы обращаться к файлам из хост-узла;
- монтировать внешнее хранилище в томе для обеспечения непрерывного хранения данных модуля во всех перезапусках модуля;
- отделять модуль от инфраструктуры хранения с помощью ресурсов PersistentVolume и PersistentVolumeClaim;
- получать ресурсы PersistentVolume требуемого (или установленного по умолчанию) класса хранилища, динамически резервируемого для каждого запроса PersistentVolumeClaim;
- предотвращать вмешательство динамического поставщика, когда вам требуется, чтобы заявка PersistentVolumeClaim была связана с предварительно зарезервированным ресурсом PersistentVolume.

В следующей главе вы увидите, какие механизмы система Kubernetes дает для предоставления данных конфигурации, информации о секретах и метаданных о модуле и контейнере процессам, запущенным в модуле. Это делается с помощью томов специальных типов, которые мы упомянули в этой главе, но пока еще не исследовали.

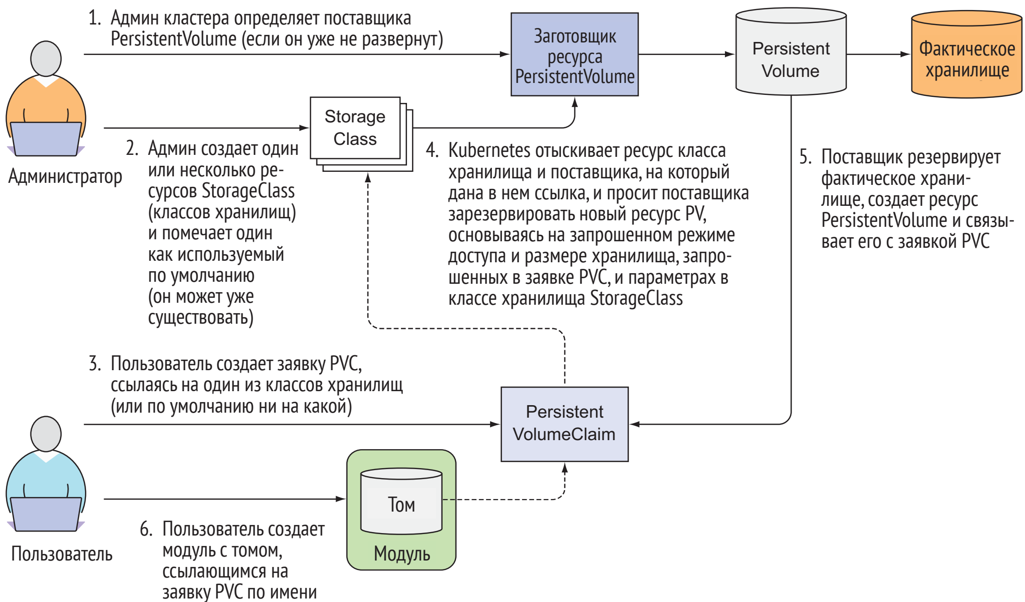


Рис. 6.10. Полная картина динамического резервирования ресурсов PersistentVolume

Глава 7

Словари конфигурации (ConfigMap) и секреты (Secret): настройка приложений

Эта глава посвящена:

- изменению главного процесса контейнера;
- передаче параметров командной строки в приложение;
- установке переменных среды, доступных для приложения;
- настройке приложения посредством словарей конфигурации `ConfigMap`;
- передаче чувствительной информации посредством секретов `Secret`.

До сего момента вам не приходилось передавать какие-либо конфигурационные данные в приложения, которые вы выполняли в упражнениях в этой книге. Поскольку почти все приложения требуют конфигурирования (параметры, которые различаются от одного развернутого экземпляра к другому, учетные данные для доступа к внешним системам и т. д.), которое не должно быть зашито в само собранное приложение, давайте посмотрим, каким образом в Kubernetes можно передавать параметры конфигурации в ваше приложение во время его запуска.

7.1 Конфигурирование контейнерных приложений

Прежде чем мы займемся рассмотрением того, как передавать конфигурационные данные в приложения, работающие в Kubernetes, давайте взглянем на то, как контейнерезированные приложения обычно конфигурируются.

Если вы отложите в сторону тот факт, что вы можете впечатывать конфигурацию в само приложение, во время начала разработки нового приложения вы обычно начинаете с того, что приложение настраивается с помощью аргументов командной строки. Затем, по мере того как список параметров растет, вы перемещаете конфигурацию в файл config.

Еще один способ передачи параметров конфигурации в приложение, который находит широкое распространение в контейнеризированных приложениях, – использовать переменные среды. Вместо того чтобы читать конфигурационный файл или аргументы командной строки, приложение отыскивает значение определенной переменной среды. В официальном образе контейнера MySQL, например, используется переменная среды под названием `MYSQL_ROOT_PASSWORD`, которая служит для установки пароля для учетной записи суперпользователя root.

Но почему переменные среды так популярны в контейнерах? Использование конфигурационных файлов внутри контейнеров Docker немного сложнее, так как вам придется впечатывать конфигурационный файл в сам образ контейнера или монтировать в контейнере том, содержащий этот файл. Очевидно, что впечатывание файлов в образ похоже на жесткое кодирование конфигурации в исходном коде приложения, потому что это требует, чтобы вы выполняли повторную сборку образа всякий раз, когда вы хотите изменить конфигурацию. Кроме того, все, у кого есть доступ к образу, могут увидеть конфигурацию, включая любую информацию, которая должна храниться в секрете, такую как учетные данные или ключи шифрования. Использовать тома – лучше, но все же требует, чтобы файл был записан в том до запуска контейнера.

Если вы читали предыдущую главу, то можете подумать об использовании тома `gitRepo` в качестве источника конфигурации. Это не плохая идея, потому что она позволяет хранить конфигурацию корректно версионированной и дает возможность легко откатить изменение конфигурации, если это необходимо. Но более простой способ позволяет поместить конфигурационные данные в ресурс верхнего уровня Kubernetes и сохранить его и все другие определения ресурсов в том же репозитории Git или в любом другом файловом хранилище. Ресурс Kubernetes для хранения конфигурационных данных называется словарем конфигурации ConfigMap. И в этой главе мы узнаем, как им пользоваться.

Независимо от того, используете вы для хранения конфигурационных данных словарь конфигурации ConfigMap или нет, вы можете настроить свои приложения с помощью:

- передачи в контейнеры аргументов командной строки;
- настройки своих собственных переменных среды для каждого контейнера;
- монтирования конфигурационных файлов в контейнеры через специальный тип тома.

Мы рассмотрим все эти варианты в следующих нескольких разделах, но, прежде чем мы начнем, давайте рассмотрим параметры конфигурации с точки зрения безопасности. Хотя многие параметры конфигурации чувствительной информации не содержат, некоторые из них могут. К ним относятся учетные данные, закрытые ключи шифрования и аналогичные данные, которые должны быть защищены. Этот тип информации должен обрабатываться с особой осторожностью, поэтому Kubernetes предлагает еще один тип объекта первого класса, который называется секретом (Secret). Мы познакомимся с ним в последней части этой главы.

7.2 Передача в контейнеры аргументов командной строки

До этого во всех примерах вы создавали контейнеры, которые выполняли команду по умолчанию, определенную в образе контейнера, но Kubernetes позволяет переопределять эту команду в рамках определения контейнера модуля, если нужно запустить другой исполняемый файл вместо указанного в образе либо требуется запустить его с другим набором аргументов командной строки. Сейчас мы посмотрим, как это сделать.

7.2.1 Определение команды и аргументов в Docker

Первое, что нужно объяснить, – это то, что в целом команда, которая выполняется в контейнере, состоит из двух частей: *команд* и *аргументов*.

Инструкции ENTRYPOINT и CMD

В файле Dockerfile две инструкции определяют две части:

- `ENTRYPOINT` определяет исполняемый файл, вызываемый при запуске контейнера;
- `CMD` задает аргументы, которые передаются в точку входа.

Хотя для указания команды, которую требуется выполнить при запуске образа, можно использовать инструкцию `CMD`, правильный способ – сделать это с помощью инструкции `ENTRYPOINT` и указать `CMD` только в том случае, если требуется определить аргументы по умолчанию. После этого образ можно запустить без указания аргументов:

```
$ docker run <образ>
```

или с дополнительными аргументами, которые переопределяют в файле Dockerfile все то, что задано в инструкции `CMD`:

```
$ docker run <образ> <аргументы>
```

Разница между формами shell и exec

Но есть еще кое-что. Обе инструкции поддерживают две разные формы:

- форма `shell` – например, `ENTRYPOINT node app.js`;
- форма `exec` – например, `ENTRYPOINT ["node", "app.js"]`.

Разница между ними заключается в том, вызывается указанная команда внутри оболочки или нет.

В образе `kubia`, созданном в главе 2, использовалась форма `exec` инструкции `ENTRYPOINT`:

```
ENTRYPOINT ["node", "app.js"]
```

Она выполняет процесс узла непосредственно (не внутри оболочки), как можно увидеть, выведя список процессов, запущенных внутри контейнера:

```
$ docker exec 4675d ps x
PID TTY    STAT   TIME   COMMAND
1 ?        Ssl    0:00   node app.js
12 ?       Rs     0:00   ps x
```

Если бы вы использовали форму `shell` (`ENTRYPOINT node app.js`), то это были бы процессы контейнера:

```
$ docker exec -it e4bad ps x
PID TTY    STAT   TIME   COMMAND
1 ?        Ss     0:00   /bin/sh -c node app.js
7 ?        Sl     0:00   node app.js
13 ?       Rs+    0:00   ps x
```

Как видите, в этом случае главным процессом (`PID 1`) будет процесс `shell` вместо процесса узла. Процесс узла (`PID 7`) запустится из процесса `shell`. Процесс `shell` не нужен, поэтому всегда следует использовать форму `exec` инструкции `ENTRYPOINT`.

Возможность настройки интервала в образе fortune

Давайте модифицируем ваш сценарий и образ fortune, так чтобы интервал задержки в цикле был настраиваемым. Вы добавите переменную `INTERVAL` и инициализируете ее значением первого аргумента командной строки, как показано в следующем ниже листинге.

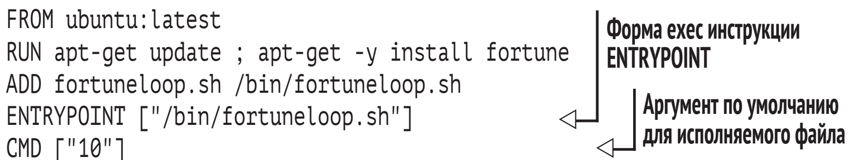
Листинг 7.1. Сценарий fortune с интервалом, настраиваемым через аргумент: fortune-args/fortuneloop.sh

```
#!/bin/bash
trap "exit" SIGINT
INTERVAL=$1
echo Configured to generate new fortune every $INTERVAL seconds
```

```
mkdir -p /var/htdocs
while :
do
  echo $(date) Writing fortune to /var/htdocs/index.html
  /usr/games/fortune > /var/htdocs/index.html
  sleep $INTERVAL
done
```

Вы добавили или модифицировали строки, выделенные жирным шрифтом. Теперь вы измените файл Dockerfile так, чтобы он использовал версию `exec` инструкции `ENTRYPOINT` и задавал интервал по умолчанию, равный 10 секундам, с помощью инструкции `CMD`, как показано в следующем ниже листинге.

Листинг 7.2. Файл Dockerfile для обновленного образа fortune: fortune-args/Dockerfile




```
FROM ubuntu:latest
RUN apt-get update ; apt-get -y install fortune
ADD fortuneloop.sh /bin/fortuneloop.sh
ENTRYPOINT ["/bin/fortuneloop.sh"]
CMD ["10"]
```

Теперь можно создать образ и отправить его в Docker Hub. На этот раз вы протегируете образ как `args` вместо `latest`:

```
$ docker build -t docker.io/luksa/fortune:args .
$ docker push docker.io/luksa/fortune:args
```

Вы можете проверить образ локально с помощью Docker:

```
$ docker run -it docker.io/luksa/fortune:args
Configured to generate new fortune every 10 seconds
Fri May 19 10:39:44 UTC 2017 Writing fortune to /var/htdocs/index.html
```

> **ПРИМЕЧАНИЕ.** Этот сценарий можно остановить с помощью комбинации клавиш Ctrl+C.

И вы можете переопределить интервал сна, выбранный по умолчанию, передав его в качестве аргумента:

```
$ docker run -it docker.io/luksa/fortune:args 15
Configured to generate new fortune every 15 seconds
```

Теперь, когда вы уверены, что ваш образ принимает переданный ему аргумент, давайте посмотрим, как его использовать в модуле.

7.2.2 Переопределение команды и аргументов в Kubernetes

В Kubernetes при указании контейнера можно переопределять и инструкцию `ENTRYPOINT`, и инструкцию `CMD`. Для этого задайте свойства `command` и

`args` в спецификации контейнера, как показано в следующем ниже листинге.

Листинг 7.3. Определение модуля, задающее пользовательскую команду и аргументы

```
kind: Pod
spec:
  containers:
  - image: some/image
    command: ["/bin/command"]
    args: ["arg1", "arg2", "arg3"]
```

В большинстве случаев вы задаете только пользовательские аргументы и редко переопределяете команду (за исключением универсальных образов, таких как busybox, в которых точка входа вообще не определена).

ПРИМЕЧАНИЕ. После создания модуля поля command и args не обновляются.

Две инструкции файла Dockerfile и эквивалентные поля секции `spec` модуля показаны в табл. 7.1.

Таблица 7.1. Определение исполняемого файла и его аргументов в Docker по сравнению с Kubernetes

Docker	Kubernetes	Описание
`ENTRYPOINT`	`command`	Исполняемый файл, который исполняется внутри контейнера
`CMD`	`args`	Аргументы, передаваемые в исполняемый файл

Запуск модуля fortune с индивидуально настроенным интервалом

Для того чтобы запустить модуль fortune с индивидуально настроенным интервалом задержки, вы скопируете свой файл fortune-pod.yaml в файл fortune-pod-args.yaml и измените его, как показано в следующем ниже листинге.

Листинг 7.4. Передача аргумента в определении модуля: fortune-pod-args.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune2s              <- Вы изменили имя модуля
spec:
  containers:
  - image: luksa/fortune:args  <- Использование fortune:args вместо fortune:latest
    args: ["2"]                <- Этот аргумент заставляет сценарий генерировать новую цитату каждые две секунды
    name: html-generator
    volumeMounts:
```

```
    - name: html
      mountPath: /var/htdocs
...
```

В определение контейнера вы добавили массив `args`. Теперь попробуйте создать этот модуль. Значения массива будут переданы в контейнер в качестве аргументов командной строки при его запуске.

Обозначение массива, которое используется в этом листинге, неплохо подходит для ситуаций, когда у вас мало аргументов. Если их несколько, то также можно использовать следующую форму записи:

```
args:
- foo
- bar
- "15"
```

> **СОВЕТ.** Заключать в кавычки строковые значения не нужно (но вот заключать в кавычки числа обязательно).

Указание аргументов – это один из способов передачи параметров конфигурации в контейнеры через аргументы командной строки. Далее вы увидите, как это делается с помощью переменных среды.

7.3 Настройка переменных среды для контейнера

Как уже отмечалось, в контейнеризированных приложениях часто в качестве источника параметров конфигурации используются переменные среды. Kubernetes позволяет задавать свой собственный список переменных среды для каждого контейнера модуля. Это показано на рис. 7.1. Хотя было бы полезно также определять переменные среды на уровне модуля и делать их наследуемыми его контейнерами, в настоящее время такой возможности не существует.

> **ПРИМЕЧАНИЕ.** Так же, как и команда и аргументы контейнера, список переменных окружения после создания модуля не обновляется.

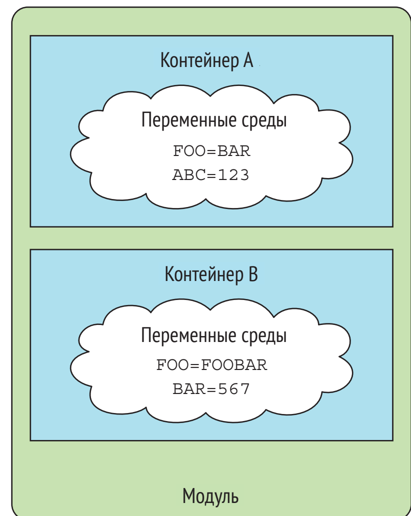


Рис. 7.1. Переменные среды могут быть заданы для каждого контейнера

Возможность настройки интервала в образе fortune с помощью переменной среды

Давайте посмотрим, как изменить ваш сценарий fortuneloop.sh еще раз, чтобы он мог быть настроен из переменной среды, как показано в следующем ниже листинге.

Листинг 7.5. Сценарий fortune с интервалом, настраиваемым через переменную среды: fortune-env/fortuneloop.sh

```
#!/bin/bash
trap "exit" SIGINT
echo Configured to generate new fortune every $INTERVAL seconds
mkdir -p /var/htdocs
while :
do
  echo $(date) Writing fortune to /var/htdocs/index.html
  /usr/games/fortune > /var/htdocs/index.html
  sleep $INTERVAL
done
```

Вам нужно было только удалить строку, в которой инициализируется переменная `INTERVAL`. Поскольку вашим «приложением» является простой скрипт bash, вам не нужно было ничего делать. Если бы приложение было написано на Java, то вы бы использовали `System.getenv("INTERVAL")`, тогда как в Node.js вы бы использовали `process.env.INTERVAL`, и в Python вы будете использовать `os.environ['INTERVAL']`.

7.3.1 Указание переменных среды в определении контейнера

После создания нового образа (на этот раз я пометил его как `luksa/fortune:env`) и отправки его в Docker Hub вы можете запустить его, создав новый модуль, в котором вы передаете переменную среды сценарию, включив ее в определение контейнера, как показано в следующем ниже листинге.

Листинг 7.6. Определение переменной среды в модуле: fortune-pod-env.yaml

```
kind: Pod
spec:
  containers:
  - image: luksa/fortune:env
    env:
    - name: INTERVAL
      value: "30"
    name: html-generator
...
```

Добавление единственной переменной в список переменных среды

Как отмечалось ранее, переменная среды устанавливается внутри определения контейнера, а не на уровне модуля.

ПРИМЕЧАНИЕ. Не забывайте, что, кроме того, в каждом контейнере система Kubernetes автоматически предоставляет доступ к переменным среды для каждой службы в том же пространстве имен. Эти переменные среды, в сущности, являются автоматически внедряемой конфигурацией.

7.3.2 Ссылка на другие переменные среды в значении переменной

В предыдущем примере вы задали фиксированное значение переменной среды, но вы также можете ссылаться на ранее заданные переменные среды или любые другие существующие переменные с помощью синтаксической конструкции `$(VAR)`. Как показано в следующем ниже листинге, если вы определяете две переменные среды, вторая может включать значение первой.

Листинг 7.7. Ссылка на переменную среды внутри другой

```
env:
- name: FIRST_VAR
  value: "foo"
- name: SECOND_VAR
  value: "$(FIRST_VAR)bar"
```

В этом случае значением `SECOND_VAR` будет `"foobar"`. Аналогично этому атрибуты `command` и `args`, о которых вы узнали в разделе 7.2, могут ссылаться на переменные среды таким же образом. Вы воспользуетесь этим методом в разделе 7.4.5.

7.3.3 Отрицательная сторона жесткого кодирования переменных среды

Практически жестко закодируемые значения в определении модуля означают, что вам нужно иметь отдельные определения модуля для ваших модулей рабочего окружения и модулей окружения разработки. Для того чтобы повторно использовать одно и то же определение модуля в нескольких средах, имеет смысл отделить конфигурацию от описания модуля. К счастью, вы можете сделать это, используя словарь конфигурации ConfigMap. Он применяется в качестве источника значений переменных среды, используя поле `valueFrom` вместо поля `value`. Вы узнаете об этом далее.

7.4 Отсоединение конфигурации с помощью словаря конфигурации ConfigMap

Весь смысл конфигурации приложения заключается в том, чтобы сохранить параметры конфигурации, которые различаются между средами или часто изменяются, отдельно от исходного кода приложения. Если вы рассматриваете описание модуля как исходный код вашего приложения (и в микросервисных архитектурах это то, чем оно является на самом деле, потому что оно определяет, как собирать отдельные компоненты в функционирующую систему), ясно, что вы должны вывести конфигурацию за пределы описания модуля.

7.4.1 Знакомство со словарями конфигурции

Kubernetes позволяет выделить параметры конфигурации в отдельный объект, называемый ConfigMap, который представляет собой ассоциативный массив, содержащий пары ключ-значение, где значения варьируются от коротких литералов до полных файлов конфигурации.

Приложению не нужно читать словарь конфигурации напрямую или даже знать, что он существует. Содержимое словаря передается в контейнеры либо как переменные среды, либо как файлы в томе (см. рис. 7.2). И поскольку на переменные среды можно ссылаться в аргументах командной строки с помощью синтаксической конструкции `$(ENV_VAR)`, то вы также можете передавать записи словаря конфигурации процессам в качестве аргументов командной строки.

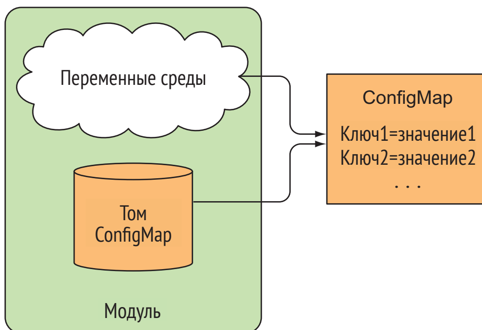


Рис. 7.2. Модули используют словари ConfigMap через переменные окружения и тома configMap

Безусловно, приложение может также прочитать содержимое словаря конфигурации непосредственно через конечную точку API REST Kubernetes, если это необходимо, но если у вас нет в этом реальной потребности, то вы должны максимально сохранить ваше приложение платформенно-независимым от Kubernetes.

Независимо от того, как приложение использует словарь конфигурации, наличие конфигурации в таком отдельном автономном объекте позволяет хранить несколько манифестов для словарей ConfigMap с тем же именем, каждый для другой среды (разработка, тестирование, контроль качества, рабочее

окружение и т. д.). Поскольку модули ссылаются на словарь ConfigMap по имени, вы можете применять другую конфигурацию в каждой среде, во всех них используя одну и ту же спецификацию модуля (см. рис. 7.3).

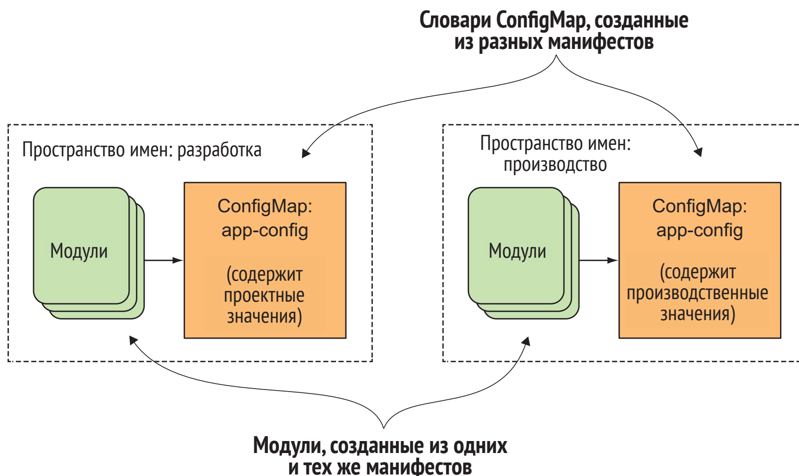


Рис. 7.3. Две разные конфигурации с одинаковыми именами, используемые в разных средах

7.4.2 Создание словаря конфигурации

Давайте посмотрим, как использовать словарь ConfigMap в одном из ваших модулей. Чтобы начать с простейшего примера, сначала создайте массив с одним ключом и примените его для заполнения переменной среды `INTERVAL` из предыдущего примера. Вместо отправки YAML с помощью универсальной команды `kubectl create -f` вы создадите словарь ConfigMap с помощью специальной команды `kubectl create configmap`.

Использование команды kubectl create configmap

Записи словаря ConfigMap можно определить, передав литералы команде `kubectl`, либо можно его создать из файлов, хранящихся на диске. Сначала используйте простой литерал:

```
$ kubectl create configmap fortune-config --from-literal=sleep-interval=25
configmap "fortune-config" created
```

> **ПРИМЕЧАНИЕ.** Ключи словаря конфигурации должны быть валидным именем поддомена (они могут содержать только буквенно-цифровые символы, дефисы, символы подчеркивания и точки). Они могут опционально содержать ведущую точку.

Ниже показано, как создается словарь конфигурации под названием `fortune-config` с одной записью `sleep-interval=25` (рис. 7.4).

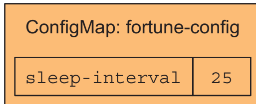


Рис. 7.4. Словарь ConfigMap sleep-interval, содержащий одну запись

Словари конфигурации, как правило, содержат более одной записи. Для того чтобы создать словарь конфигурации с несколькими литеральными записями, нужно добавить несколько аргументов `--from-literal`:

```
$ kubectl create configmap myconfigmap
➥  --from-literal=foo=bar --from-literal=bar=baz --from-literal=one=two
```

Давайте проинспектируем дескриптор YAML словаря конфигурации, который вы создали с помощью команды `kubectl get`, как показано в следующем ниже листинге.

Листинг 7.8. Определение словаря ConfigMap

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  sleep-interval: "25"          <- Единственная запись в словаре
kind: ConfigMap                 <- Этот дескриптор описывает словарь ConfigMap
metadata:
  creationTimestamp: 2016-08-11T20:31:08Z
  name: fortune-config          <- Имя этого словаря (вы ссылаетесь на него по этому имени)
  namespace: default
  resourceVersion: "910025"
  selfLink: /api/v1/namespaces/default/configmaps/fortune-config
  uid: 88c4167e-6002-11e6-a50d-42010af00237
```

Ничего особенного. Вы могли бы легко написать этот YAML самостоятельно (вам не нужно было указывать ничего, кроме имени в разделе метаданных, конечно) и отправить его в API Kubernetes с помощью известной команды

```
$ kubectl create -f fortune-config.yaml
```

Создание записи словаря конфигурации из содержимого файла

Словари ConfigMap также могут хранить сложные конфигурационные данные, такие как полные файлы конфигурации. Для этого команда `kubectl create configmap` также поддерживает чтение файлов с диска и их сохранение в виде отдельных записей в словаре конфигурации:

```
$ kubectl create configmap my-config --from-file=config-file.conf
```

При выполнении приведенной выше команды `kubectl` ищет файл `config-file.conf` в каталоге, в котором вы запустили `kubectl`. Затем он сохранит со-

держимое файла с ключом config-file.conf в словарь конфигурации (имя файла используется в качестве ключа словаря), но вы также можете указать ключ вручную, как здесь:

```
$ kubectl create configmap my-config --from-file=customkey=config-file.conf
```

Эта команда сохранит содержимое файла в ключе customkey. Как и в случае с литералами, можно добавить несколько файлов, несколько раз воспользовавшись аргументом --from-file.

Создание словаря конфигурации из файлов в каталоге

Вместо импорта каждого файла по отдельности можно даже импортировать все файлы из каталога файлов:

```
$ kubectl create configmap my-config --from-file=/path/to/dir
```

В этом случае kubectl создаст отдельную запись словаря для каждого файла в указанном каталоге, но только для файлов, имя которых является допустимым ключом словаря конфигурации.

Объединение разных вариантов

При создании словарей конфигурации можно использовать комбинацию всех упомянутых здесь вариантов (обратите внимание, что эти файлы не включены в архив кода книги, – если хотите попробовать команду, то вы можете создать их самостоятельно):

```
$ kubectl create configmap my-config
➥  --from-file=foo.json                 <— Один файл
➥  --from-file=bar=foobar.conf           <— Файл, хранящийся под собственным ключом
➥  --from-file=config-opts/             <— Весь каталог
➥  --from-literal=some=thing              <— Литеральное значение
```

Здесь вы создали словарь конфигурации из нескольких источников: целого каталога, файла, еще одного файла (сохраненного под собственным ключом вместо использования имени файла в качестве ключа) и литерального значения. Рисунок 7.5 показывает все эти источники и результирующий словарь конфигурации.

7.4.3 Передача записи словаря конфигурации в контейнер в качестве переменной среды

Как теперь подать значения из этого словаря в контейнер модуля? У вас есть три варианта. Начнем с простейшего – установки переменной среды. Вы будете использовать поле valueFrom, которое я упомянул в разделе 7.3.3. Описание модуля должно выглядеть как в листинге 7.9.

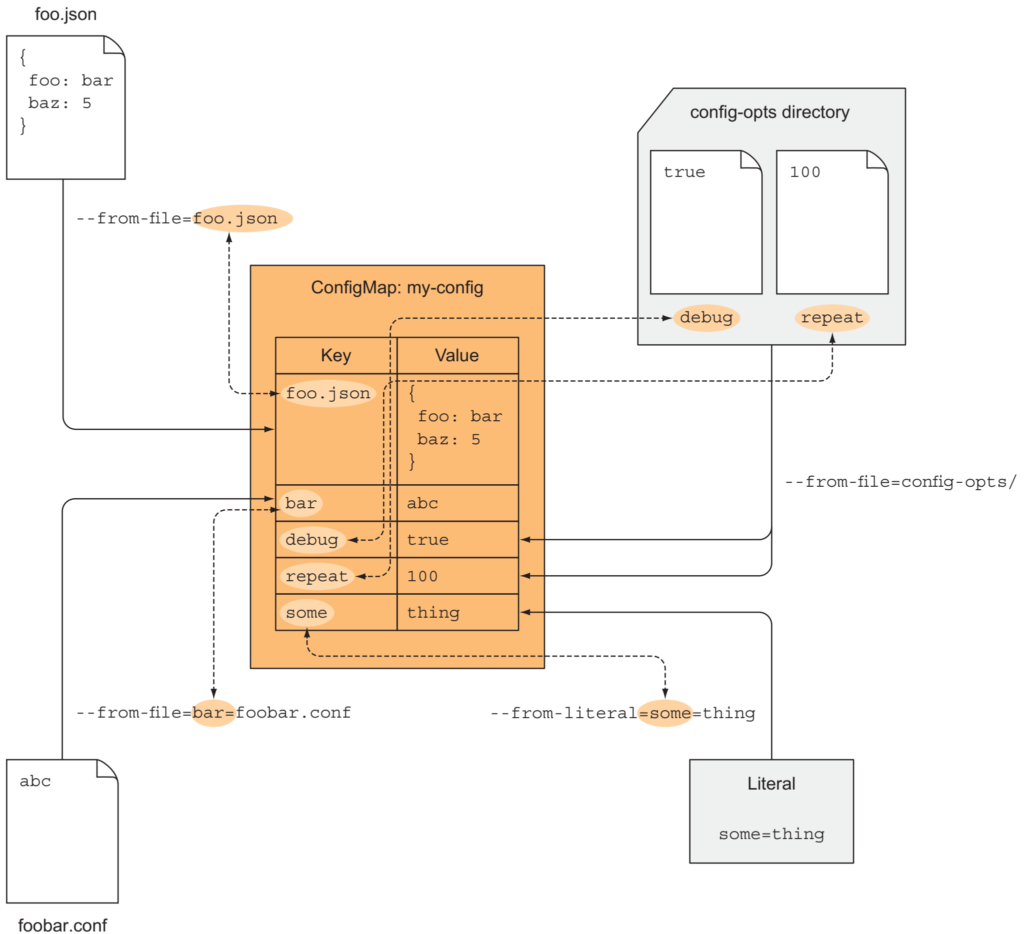


Рис. 7.5. Создание словаря ConfigMap из отдельных файлов, каталога и литерального значения

Листинг 7.9. Код с переменной среды в поле env из словаря ConfigMap: fortune-pod-env-configmap.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-env-from-configmap
spec:
  containers:
  - image: luksa/fortune:env
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: fortune-config
```

Вы устанавливаете переменную среды INTERVAL

Вместо установки фиксированного значения вы инициализируете его из ключа словаря ConfigMap

Имя словаря ConfigMap, на который вы ссылаетесь

```
        key: sleep-interval    ◁ Вы устанавливаете переменную в то, что
...                              хранится под этим ключом в словаре ConfigMap
```

Вы определили переменную среды под названием `INTERVAL` и установили ее значение на то, что хранится в словаре конфигурации `fortune-config` под ключом `sleep-interval`. Когда процесс, запущенный в контейнере html-генератора, считывает переменную среды `INTERVAL`, он увидит значение `25` (показано на рис. 7.6).

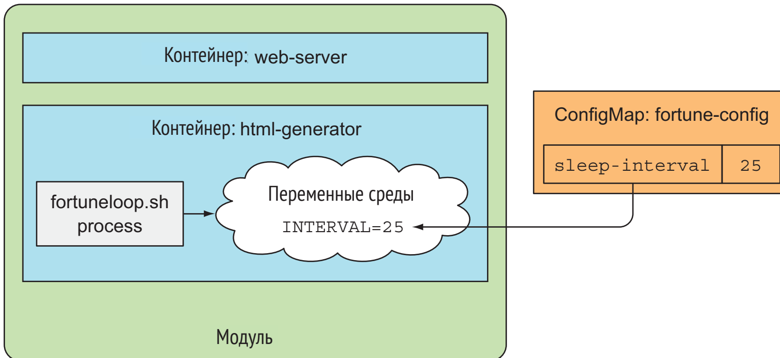


Рис. 7.6. Передача записи словаря ConfigMap в качестве переменной среды в контейнер

Ссылки на несуществующие словари конфигурации в модуле

Вы можете задаться вопросом, что произойдет, если при создании модуля указанный словарь конфигурации не существует. Kubernetes назначает модуль в нормальном режиме и пытается запустить его контейнеры. Контейнер, ссылающийся на несуществующий словарь конфигурации, запустить не удастся, а другой контейнер запустится нормально. Если вы затем создадите недостающий словарь конфигурации, то неисправный контейнер запустится, не требуя от вас воссоздания модуля.

> **ПРИМЕЧАНИЕ.** Вы также можете пометить ссылку на словарь конфигурации как необязательный (установив `configMapKeyRef.optional: true`). В этом случае контейнер запускается, даже если словарь конфигурации не существует.

В этом примере показано, как отделить конфигурацию от спецификации модуля. Это позволяет вам держать все параметры конфигурации в одном месте (даже для нескольких модулей), вместо того чтобы разбрасывать их по всему определению модуля (или дублировать по нескольким манифестам модуля).

7.4.4 Одновременная передача всех записей словаря конфигурации как переменных среды

Когда словарь конфигурации содержит больше, чем несколько записей, создание переменных среды из каждой записи по отдельности становится

утомительным и подвержено ошибкам. К счастью, Kubernetes версии 1.6 позволяет обеспечивать доступ ко всем записям словаря конфигурации как переменным среды.

Представьте, что у вас словарь конфигурации с тремя ключами под названием FOO, BAR и FOO-BAR. Все они могут быть представлены как переменные среды с помощью атрибута envFrom, а не env, как в предыдущих примерах. Пример приведен в следующем ниже листинге.

Листинг 7.10. Модуль с переменными среды, состоящий из всех записей словаря ConfigMap

```
spec:
  containers:
  - image: some-image
    envFrom:
    - prefix: CONFIG_
      configMapRef:
        name: my-config-map
...
```

Использование поля envFrom вместо поля env

Все переменные среды будут иметь префикс CONFIG_

Ссылка на словарь ConfigMap под названием my-config-map

Как можно увидеть, для переменных среды также можно указать префикс (в данном случае CONFIG_). В результате в контейнере присутствуют следующие две переменные среды: CONFIG_FOO и CONFIG_BAR.

> **ПРИМЕЧАНИЕ.** Префикс является необязательным, поэтому, если его опустить, переменные среды будут иметь то же имя, что и ключи.

Вы заметили, что я упомянул о двух переменных, но ранее я сказал, что словарь конфигурации имеет три записи (FOO, BAR и FOO-BAR)? Почему нет переменной среды для записи FOO-BAR словаря конфигурации?

Причина в том, что CONFIG_FOO-BAR не является допустимым именем переменной среды, так как содержит тире. Система Kubernetes не преобразует ключи каким-либо образом (например, не преобразует тире в подчеркивание). Если ключ словаря конфигурации не имеет правильного формата, она пропускает запись (но записывает событие, сообщающее, что она его пропустила).

7.4.5 Передача записи словаря конфигурации в качестве аргумента командной строки

Теперь давайте также рассмотрим, как передавать значения из словаря ConfigMap в качестве аргументов главному процессу, запущенному в контейнере. Как показано на рис. 7.7, нельзя ссылаться на записи словаря конфигурации непосредственно в поле pod.spec.containers.args, но можно сначала инициализировать переменную среды из записи словаря конфигурации и затем обратиться к переменной внутри аргументов.

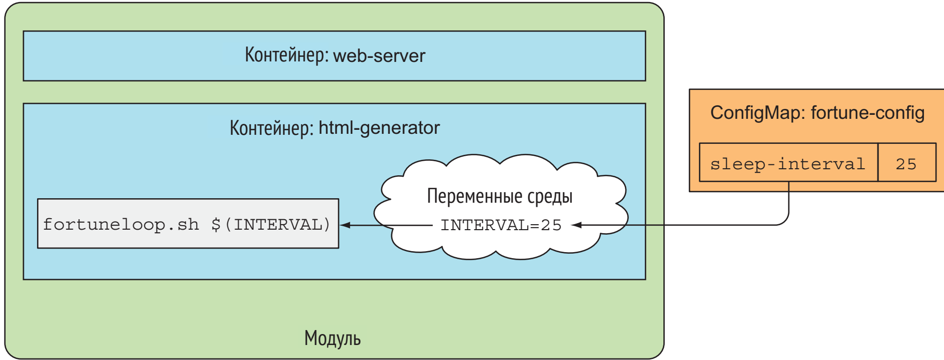


Рис. 7.7. Передача записи словаря ConfigMap в качестве аргумента командной строки

В листинге 7.11 показан пример того, как это сделать в YAML.

Листинг 7.11. Использование записи словаря ConfigMap в качестве аргументов: fortune-pod-args-configmap.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-args-from-configmap
spec:
  containers:
  - image: luksa/fortune:args          ◁ Использование образа, который принимает интервал от первого аргумента, а не от переменной среды
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:               ◁ Определение переменной среды точно так же, как и раньше
          name: fortune-config
          key: sleep-interval
    args: ["$(INTERVAL)"]              ◁ Ссылка на переменную среды в аргументе
...
```

Вы определили переменную среды точно так же, как и раньше, но затем использовали синтаксическую конструкцию `$(ENV_VARIABLE_NAME)`, чтобы Kubernetes внедрил значение переменной в аргумент.

7.4.6 Использование тома configMap для обеспечения доступа к записям словаря конфигурации в виде файлов

Передача параметров конфигурации в качестве переменных среды или аргументов командной строки обычно используется для коротких значений переменных. Словарь конфигурации, как вы убедились, также может содержать целые конфигурационные файлы. Если вы хотите предоставить их контейнеру, вы можете использовать один из специальных типов томов, о которых я упоминал в предыдущей главе, а именно том configMap.

Том `configMap` будет предоставлять доступ к каждой записи словаря конфигурации как файл. Процесс, запущенный в контейнере, может получить значение записи, прочитав содержимое файла.

Хотя этот метод в основном предназначен для передачи больших конфигурационных файлов в контейнер, ничто не мешает вам таким образом передавать короткие одиночные значения.

Создание словаря конфигурации

Вместо повторного модифицирования своего сценария fortuneloop.sh теперь вы попробуете другой пример. Вы будете использовать конфигурационный файл для настройки веб-сервера Nginx, работающего внутри контейнера веб-сервера модуля fortune. Предположим, вы хотите, чтобы ваш веб-сервер Nginx сжимал ответы, которые он отправляет клиенту. Чтобы активировать сжатие, конфигурационный файл для Nginx должен выглядеть следующим образом.

Листинг 7.12. Конфигурация Nginx с активированным сжатием gzip: my-nginx-config.conf

```
server {
  listen              80;
  server_name         www.kubia-example.com;

  gzip on;
  gzip_types text/plain application/xml;

  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
  }
}
```

Это активирует сжатие gzip для обычного текста и XML-файлов

Теперь удалите существующий словарь конфигурации `fortune-config` с помощью команды `kubectl delete configmap fortune-config`, чтобы вы могли заменить его на новый, который будет включать конфигурационный файл Nginx. Вы создадите словарь конфигурации из файлов, хранящихся на локальном диске.

Создайте новую папку с именем configmap-files и сохраните конфигурационный файл Nginx из предыдущего листинга в configmap-files/my-nginx-config.conf. Для того чтобы словарь конфигурации также содержал запись `sleep-interval`, добавьте текстовый файл sleep-interval в тот же каталог и сохраните в нем число 25 (см. рис. 7.8).

Теперь создайте словарь конфигурации из всех файлов в каталоге, как показано ниже:

```
$ kubectl create configmap fortune-config --from-file=configmap-files
configmap "fortune-config" created
```

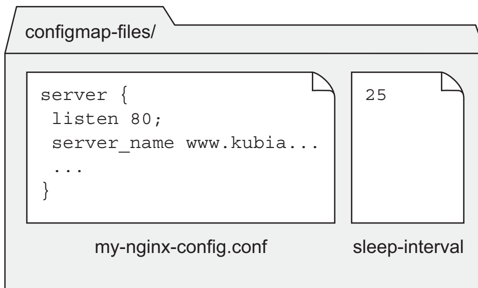


Рис. 7.8. Содержимое каталога configmap-files и его файлов

В следующем ниже листинге показано, как выглядит YAML этого словаря конфигурации.

Листинг 7.13. Определение YAML словаря ConfigMap, созданного из файла

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  my-nginx-config.conf: |
    server {
      listen              80;
      server_name         www.kubia-example.com;

      gzip on;
      gzip_types text/plain application/xml;

      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }
    }
  sleep-interval: |
    25
kind: ConfigMap
...
```

Запись, содержащая содержимое файла конфигурации Nginx

Запись sleep-interval

ПРИМЕЧАНИЕ. Вертикальная черта после двоеточия в первой строке обеих записей сигнализирует о том, что далее следует литеральное многострочное значение.

Словарь конфигурации содержит две записи с ключами, соответствующими фактическим именам файлов, из которых они были созданы. Теперь вы будете использовать словарь конфигурации в обоих контейнерах модуля.

Использование записей словаря конфигурации в томе

Создать том, заполненный содержимым словаря конфигурации, так же просто, как создать том, ссылающийся на словарь конфигурации по имени, и

смонтировать том в контейнере. Вы уже научились создавать тома и монтировать их, поэтому единственное, что осталось узнать, – это как инициализировать том файлами, созданными из записей словаря конфигурации.

Веб-сервер Nginx читает свой конфигурационный файл из /etc/nginx/nginx.conf. Образ Nginx уже содержит этот файл с параметрами конфигурации по умолчанию, которые вы не хотите переопределять, поэтому вы не будете заменять этот файл целиком. К счастью, конфигурационный файл по умолчанию автоматически также включает все файлы .conf в подкаталоге /etc/nginx/conf.d/, поэтому вы должны добавить туда свой конфигурационный файл. На рис. 7.9 показано, чего вы хотите достичь.

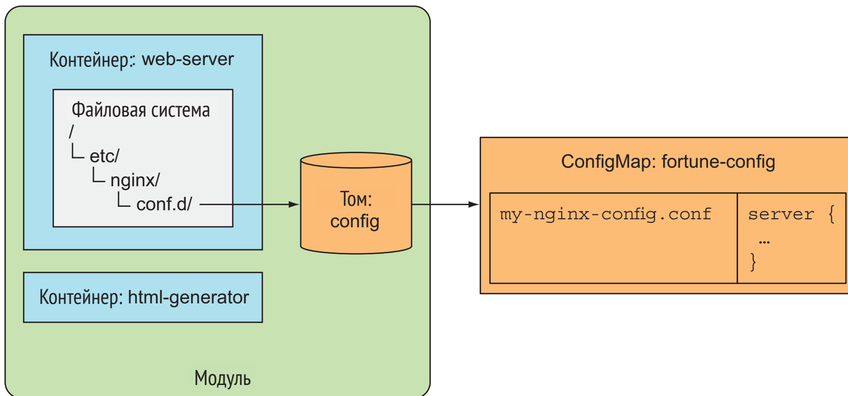


Рис. 7.9. Передача записей словаря ConfigMap в модуль в виде файлов в томе

Дескриптор модуля показан в листинге 7.14 (ненужные части опущены, но вы найдете полный файл в архиве кода).

Листинг 7.14. Модуль с записями словаря ConfigMap, смонтированными в виде файлов: fortune-pod-configmapvolume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-configmap-volume
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    ...
    - name: config
      mountPath: /etc/nginx/conf.d      <- Вы монтируете том configMap
      readOnly: true                       в это месторасположение
    ...
  volumes:
```

```
...
- name: config
  configMap:
    name: fortune-config
...
```

Том ссылается на ваш словарь ConfigMap fortune-config

Это определение модуля включает том, который ссылается на ваш словарь конфигурации `fortune-config`. Для того чтобы веб-сервер Nginx смог его использовать, вы монтируете этот том в каталог /etc/nginx/conf.d.

Проверка использования сервером Nginx смонтированного конфигурационного файла

Теперь веб-сервер должен быть настроен на сжатие отправляемых ответов. Это можно проверить, включив переадресацию портов с localhost: 8080 на порт 80 модуля – и проверив отклик сервера с помощью `curl`, как показано в следующем ниже листинге.

Листинг 7.15. Проверка включения сжатия откликов веб-сервера Nginx

```
$ kubectl port-forward fortune-configmap-volume 8080:80 &
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
$ curl -H "Accept-Encoding: gzip" -I localhost:8080
HTTP/1.1 200 OK
Server: nginx/1.11.1
Date: Thu, 18 Aug 2016 11:52:57 GMT
Content-Type: text/html
Last-Modified: Thu, 18 Aug 2016 11:52:55 GMT
Connection: keep-alive
ETag: W/"57b5a197-37"
Content-Encoding: gzip
```

Это показывает, что отклик сжат

Исследование содержимого смонтированного тома configMap

Отклик показывает, что вы достигли того, чего хотели, но давайте теперь посмотрим, что находится в каталоге /etc/nginx/conf.d:

```
$ kubectl exec fortune-configmap-volume -c web-server ls /etc/nginx/conf.d
my-nginx-config.conf
sleep-interval
```

Обе записи из словаря конфигурации были добавлены в каталог в виде файлов. Запись `sleep-interval` также включена, хотя у нее нет никакого смысла находиться там, потому что она предназначена только для использования контейнером `fortuneloop`. Можно создать два разных словаря конфигурации и использовать один для настройки контейнера `fortuneloop` и другой для настройки контейнера `web-server`. Но почему-то кажется неправильным ис-

пользовать несколько словарей конфигурации для настройки контейнеров одного и того же модуля. В конце концов, наличие контейнеров в одном модуле означает, что контейнеры тесно связаны и, вероятно, также должны быть настроены как единое целое.

Предоставление доступа к определенным записям словаря конфигурации в томе

К счастью, том configMap можно заполнить только частью записей словаря конфигурации – в вашем случае только записью my-nginx-config.conf. Это не повлияет на контейнер fortuneloop, поскольку вы передаете ему запись sleep-interval через переменную среды, а не через том.

Чтобы определить, какие записи должны быть представлены как файлы в томе configMap, используйте атрибут items тома, как показано в следующем ниже листинге.

Листинг 7.16. Модуль с конкретной записью словаря ConfigMap, установленный в файловый каталог: fortune-pod-configmap-volume-with-items.yaml

```
volumes:
- name: config
  configMap:
    name: fortune-config
    items:                          ◁ Выбор записей для включения в том путем их перечисления
    - key: my-nginx-config.conf     ◁ Вы хотите включить запись под этим ключом
      path: gzip.conf               ◁ Значение записи должно быть сохранено в этом файле
```

При указании отдельных записей необходимо задать имя файла для каждой отдельной записи вместе с ключом записи. Если вы запустите модуль из предыдущего листинга, то каталог /etc/nginx/conf.d останется в чистоте, так как он содержит только файл gzip.conf и ничего больше.

Монтирование каталога скрывает существующие файлы в этом каталоге

В этом месте следует обсудить один важный аспект. И в этом, и в предыдущем примерах вы смонтировали том в качестве каталога, то есть вы скрыли все файлы, которые хранятся в каталоге /etc/nginx/conf.d непосредственно в самом образе контейнера.

Обычно это происходит в Linux при монтировании файловой системы в непустой каталог. Тогда каталог содержит только файлы из смонтированной файловой системы, в то время как исходные файлы в этом каталоге недоступны до тех пор, пока файловая система смонтирована.

В вашем случае это не имеет ужасных побочных эффектов, но представьте, что вы монтируете том в каталог /etc, который обычно содержит много важных файлов. Скорее всего, это приведет к повреждению всего контейнера, так как все исходные файлы, которые должны находиться в каталоге /etc, больше не

будут там находиться. Если вам нужно добавить файл в каталог типа /etc, то вам вообще нельзя использовать этот метод.

Монтирование отдельных записей словаря конфигурации в виде файлов без скрытия других файлов в каталоге

Естественно, теперь вам интересно, как добавить отдельные файлы из словаря ConfigMap в существующий каталог, не скрывая хранящиеся в нем существующие файлы. Дополнительное свойство `subPath` в `volumeMount` позволяет монтировать либо один файл, либо один каталог из тома вместо монтирования всего тома. Возможно, это легче объяснить визуально (см. рис. 7.10).

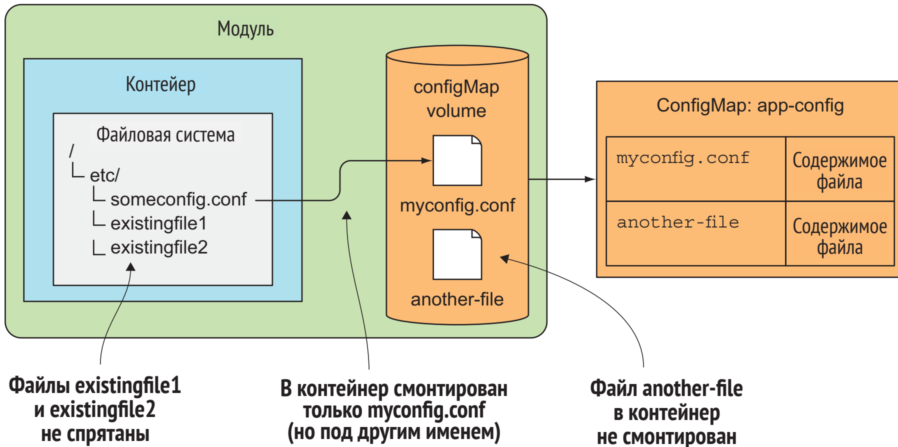


Рис. 7.10. Монтирование одного файла из тома

Предположим, имеется том configMap, содержащий файл myconfig.conf, который вы хотите добавить в каталог /etc как someconfig.conf. Вы можете задействовать свойство `subPath`, для того чтобы смонтировать его, не затрагивая другие файлы в этом каталоге. Соответствующая часть определения модуля показана в следующем ниже листинге.

Листинг 7.17. Модуль с определенной записью словаря ConfigMap, смонтированный в конкретный файл

```
spec:
  containers:
  - image: some/image
    volumeMounts:
    - name: myvolume
      mountPath: /etc/someconfig.conf    <- Вы монтируете в файл, а не в каталог
      subPath: myconfig.conf             <- Вместо монтирования всего тома целиком вы монтируете только запись myconfig.conf
```

Свойство `subPath` можно использовать при монтировании любого типа тома. Вместо монтирования всего тома можно смонтировать его часть. Но этот способ монтирования отдельных файлов имеет относительно большой

недостаток, связанный с обновлением файлов. Вы узнаете больше об этом в следующем разделе, но сначала давайте поговорим о начальном состоянии тома configMap, сказав несколько слов о разрешениях для файлов.

Настройка разрешений для файлов в томе configMap

По умолчанию разрешения для всех файлов в томе configMap имеют значение 644 (`-rw-r--r--`). Это можно изменить, установив свойство `defaultMode` в спецификации тома, как показано в следующем ниже листинге.

Листинг 7.18. Установка разрешений для файлов: fortune-pod-configmap-volume-defaultMode.yaml

```
volumes:
- name: config
  configMap:
    name: fortune-config
    defaultMode: "6600"    <-- Это устанавливает для всех файлов разрешения -rw-rw------
```

Хотя словарь конфигурации следует использовать для нечувствительных конфигурационных данных, вы можете сделать файл доступным для чтения и записи только для пользователя и группы, которые владеют этим файлом, как показано в примере, приведенном в предыдущем листинге.

7.4.7 Обновление конфигурации приложения без перезапуска приложения

Мы отметили, что одним из недостатков использования переменных среды или аргументов командной строки в качестве источника конфигурации является невозможность их обновления во время выполнения процесса. Использование словаря конфигурации и предоставление к нему доступа через том дает возможность обновлять конфигурацию без необходимости повторного создания модуля или даже перезапуска контейнера.

Когда вы обновляете словарь конфигурации, ссылающиеся на него файлы во всех томах будут обновлены. И после этого на процессе лежит обязанность обнаружить, что они изменились, и их перезагрузить. Однако, скорее всего, в Kubernetes в конечном итоге также будет принята поддержка отправки сигнала в контейнер после обновления файлов.

> **ПРЕДУПРЕЖДЕНИЕ.** Следует учесть, что в то время, когда писалась эта книга, на обновление файлов после обновления словаря конфигурации уходило удивительно много времени (это может занимать до целой минуты).

Редактирование словаря конфигурации

Давайте посмотрим, как можно изменить словарь конфигурации и дать процессу, работающему в модуле, перезагрузить файлы, представленные в

томе `configMap`. Вы измените файл nginx.config из предыдущего примера и заставите веб-сервер Nginx использовать новую конфигурацию без перезагрузки модуля. Попробовать включить сжатие gzip, отредактировав словарь конфигурации `fortune-config` с помощью команды `kubectl edit`:

```
$ kubectl edit configmap fortune-config
```

После открытия редактора измените строку `gzip on` на `gzip off`, сохраните файл и закройте редактор. Словарь конфигурации затем обновится, и вскоре после этого сам файл в томе тоже обновится. Это можно подтвердить, распечатав содержимое файла с помощью команды `kubectl exec`:

```
$ kubectl exec fortune-configmap-volume -c web-server
➥  cat /etc/nginx/conf.d/my-nginx-config.conf
```

Если вы еще не видите обновление, подождите некоторое время и повторите попытку. Для обновления файлов требуется некоторое время. В конце концов, вы увидите изменение в файле конфигурации, но вы обнаружите, что это не влияет на Nginx, потому что он не отслеживает файлы и не перезагружает их автоматически.

Подача сигнала веб-серверу Nginx для перезагрузки конфигурации

Веб-сервер Nginx будет продолжать сжимать свои ответы до тех пор, пока вы не сообщите ему о том, что надо перезагрузить свои конфигурационные файлы. Это можно сделать с помощью следующей ниже команды:

```
$ kubectl exec fortune-configmap-volume -c web-server -- nginx -s reload
```

Теперь, если вы попытаетесь снова выйти на сервер с помощью `curl`, то вы увидите, что отклик больше не сжимается (он больше не содержит заголовок `Content-Encoding: gzip`). Вы фактически изменили конфигурацию приложения без перезапуска контейнера или повторного создания модуля.

Каким образом файлы обновляются атомарно

Вы можете задаться вопросом, что произойдет, если приложение самостоятельно сможет обнаруживать изменения в конфигурационном файле и перезагружать их до того, как Kubernetes закончит обновлять все файлы в томе `configMap`. К счастью, этого не произойдет, потому что все файлы обновляются атомарно, то есть все обновления происходят сразу. Kubernetes достигает этого с помощью символических ссылок. Если вы выведете список всех файлов в смонтированном томе `configMap`, вы увидите что-то вроде следующего ниже.

Листинг 7.19. Файлы в смонтированном томе configMap

```
$ kubectl exec -it fortune-configmap-volume -c web-server -- ls -lA
➥  /etc/nginx/conf.d
```

```
total 4
drwxr-xr-x ... 12:15 ..4984_09_04_12_15_06.865837643
lrwxrwxrwx ... 12:15 ..data -> ..4984_09_04_12_15_06.865837643
lrwxrwxrwx ... 12:15 my-nginx-config.conf -> ..data/my-nginx-config.conf
lrwxrwxrwx ... 12:15 sleep-interval -> ..data/sleep-interval
```

Как вы можете видеть, файлы в смонтированном томе configMap являются символическими ссылками, указывающими на файлы в каталоге `..data`. Каталог `..data` также является символической ссылкой, указывающей на каталог `..4984_09_04_something`. Когда словарь конфигурации обновляется, Kubernetes создает новый каталог, как этот, записывает все файлы в него, а затем повторно связывает символическую ссылку `..data` с новым каталогом, практически одновременно изменяя все файлы.

Файлы, смонтированные в существующие каталоги, не обновляются

Один серьезный нюанс относится к обновлению томов, поддерживаемых словарем конфигурации. Если вы смонтировали только один файл в контейнере вместо всего тома, то этот файл не будет обновлен! По крайней мере, это верно на момент написания данной главы.

На данный момент, если вам нужно добавить отдельный файл и обновить его при обновлении исходного словаря конфигурации, одно обходное решение состоит в том, чтобы смонтировать весь том целиком в другой каталог, а затем создать символическую ссылку, указывающую на соответствующий файл. Символическая ссылка может быть либо создана в самом образе контейнера, либо вы можете создать символическую ссылку при запуске контейнера.

Последствия обновления словаря конфигурации

Одной из наиболее важных особенностей контейнеров является их неизменяемость, которая позволяет нам быть уверенными, что нет никаких различий между несколькими запущенными контейнерами, созданными из того же образа, поэтому разве не будет неправильным обходить эту неизменяемость, изменяя словарь конфигурации, используемый работающими контейнерами?

Основная проблема возникает, когда приложение не поддерживает перезагрузку конфигурации. Это приводит к тому, что различные работающие экземпляры настраиваются по-разному – те модули, которые создаются после изменения словаря конфигурации, будут использовать новую конфигурацию, в то время как старые модули по-прежнему будут использовать старую. И это не ограничивается новыми модулями. Если контейнер модуля перезапущен (по какой-либо причине), то новый процесс также увидит новую конфигурацию. Поэтому, если приложение не перезагружает свою конфигурацию автоматически, изменение существующего словаря конфигурации (в то время когда модули его используют) не может быть хорошей идеей.

Если приложение поддерживает перезагрузку конфигурации, изменение словаря конфигурации обычно не имеет большого значения, но вы должны

знать, что, поскольку файлы в томах configMap не обновляются синхронно во всех запущенных экземплярах, файлы в отдельных модулях могут быть рассинхронизированы в течение целой минуты.

7.5 Использование секретов для передачи чувствительных данных в контейнеры

Все данные, которые вы передавали своим контейнерам до сих пор, являются регулярными, нечувствительными конфигурационными данными, которые не должны храниться в безопасности. Но, как мы уже отмечали в начале главы, конфигурация обычно также содержит чувствительную информацию, такую как учетные данные и закрытые ключи шифрования, которые должны быть защищены.

7.5.1 Знакомство с секретами

Для хранения и распространения таких данных Kubernetes предоставляет отдельный объект, называемый Secret. Секреты очень похожи на словари конфигурации. Они такие же ассоциативные массивы, которые содержат пары ключ-значение. Их можно использовать так же, как словарь конфигурации. Можно:

- передавать записи секрета в контейнер в качестве переменных среды;
- предоставлять доступ к записям секрета в виде файлов в томе.

Kubernetes помогает хранить ваши секреты в безопасности, гарантируя, что каждый секрет распространяется только на узлы, управляющие модулями, которые нуждаются в доступе к секрету. Кроме того, на самих узлах секреты всегда хранятся в памяти и никогда не записываются в физическое хранилище, что потребовало бы очистки дисков после удаления оттуда секретов.

На самом главном узле (в частности, в etcd) секреты обычно хранились в незашифрованном виде, то есть ведущий узел должен обеспечивать возможность держать чувствительные данные, хранящиеся в секретах, в безопасности. Это включает не только обеспечение безопасности хранилища etcd, но и предотвращение использования сервера API неавторизованными пользователями, поскольку любой, кто может создавать модули, может смонтировать секрет Secret в модуль и через него получить доступ к чувствительным данным. Начиная с Kubernetes версии 1.7 etcd хранит секреты в зашифрованном виде, что делает систему гораздо безопаснее. По этой причине крайне важно правильно выбирать, когда использовать секрет, а когда словарь конфигурации. Выбор между ними прост:

- используйте словарь конфигурации для хранения незащищенных, простых конфигурационных данных;
- используйте секрет для хранения чувствительных данных, которые должны храниться под замком. Если конфигурационный файл содер-

жит как чувствительные, так и нечувствительные данные, то следует хранить файл в секрете Secret.

Вы уже использовали секреты в главе 5, когда создавали секрет для хранения сертификата TLS, необходимый для ресурса Ingress. Теперь вы займетесь исследованием секретов более подробно.

7.5.2 Знакомство с секретом default-token

Вы начнете с того, что исследуете секрет, который смонтирован в каждом контейнере, который вы запускаете. Вы, возможно, заметили это, когда использовали команду `kubectl describe` с модулем. Результат этой команды всегда содержал что-то вроде этого:

```
Volumes:
  default-token-cfee9:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-cfee9
```

За каждым модулем автоматически закреплен том `secret`. Этот том в приведенном выше результате команды `kubectl describe` ссылается на секрет под названием `default-token-cfee9`. Поскольку секреты являются ресурсами, их можно вывести на экран в виде списка с помощью команды `kubectl get secrets` и найти в этом списке секрет `default-token`. Давайте посмотрим:

```
$ kubectl get secrets
NAME                  TYPE                                  DATA  AGE
default-token-cfee9   kubernetes.io/service-account-token   3     39d
```

Команду `kubectl describe` можно также использовать, чтобы узнать о нем немного больше, как показано в следующем ниже листинге.

Листинг 7.20. Описание секрета

```
$ kubectl describe secrets
Name:         default-token-cfee9
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=default
              kubernetes.io/service-account.uid=cc04bb39-b53f-42010af00237
Type:         kubernetes.io/service-account-token
Data
====
ca.crt:       1139 bytes
namespace:    7 bytes
token:        eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

Этот секрет содержит три записи

Вы видите, что секрет содержит три записи, `ca.crt`, `namespace` и `token`, которые представляют все, что вам нужно для безопасного обмена с сервером

API Kubernetes из ваших модулей если вам это нужно. Хотя в идеале вы хотите, чтобы ваше приложение было полностью платформенно-независимым от Kubernetes, когда нет альтернативы, кроме как напрямую обмениваться с Kubernetes, вы будете использовать файлы, предоставленные через этот том `secret`.

Команда `kubectl describe pod` показывает, где смонтирован том `secret`:

```
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-cfee9
```

> **ПРИМЕЧАНИЕ.** По умолчанию секрет `default-token` вмонтирован в каждый контейнер, но вы можете отключить этот функционал в каждом модуле, задав значение `false` в поле `automountServiceAccountToken` в секции `spec` модуля или же в учетной записи служб, которую модуль использует. (Об учетных записях служб вы узнаете далее в книге.)

Чтобы помочь вам визуализировать, где и как смонтирован секрет `default-token`, приводим рис. 7.11.

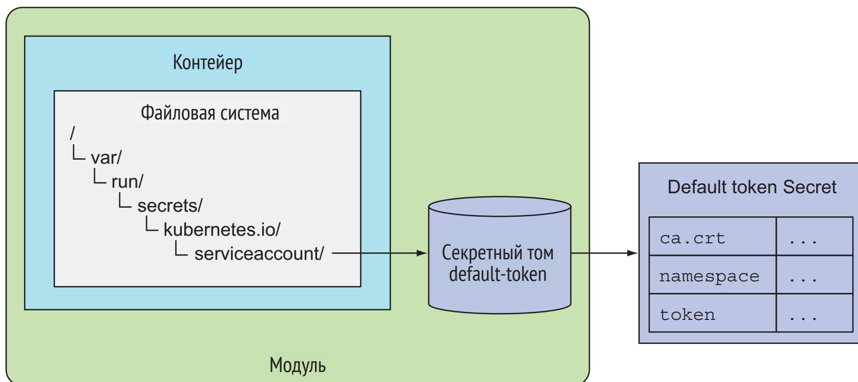


Рис. 7.11. Секрет default-token создается автоматически, и соответствующий том монтируется в каждом модуле автоматически

Мы отметили, что секреты похожи на словари конфигурации, а раз так, то поскольку этот секрет содержит три записи, вы можете ожидать, что увидите три файла в каталоге, в который смонтирован том `secret`. Это можно легко проверить с помощью команды `kubectl exec`:

```
$ kubectl exec mypod ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt
namespace
token
```

В следующей главе вы увидите, как ваше приложение может использовать эти файлы для доступа к серверу API.

7.5.3 Создание секрета

Теперь вы создадите свой маленький секрет. Вы улучшите свой контейнер с раздающим цитаты веб-сервером Nginx, настроив его для обслуживания HTTPS-трафика. Для этого необходимо создать сертификат и закрытый ключ. Закрытый ключ должен быть защищен, поэтому вы поместите его и сертификат в секрет.

Сначала создайте файлы сертификата и закрытого ключа (сделайте это на локальном компьютере). Вы также можете использовать файлы в архиве кода, прилагаемого к этой книге (файлы сертификатов и ключей находятся в каталоге fortune-https):

```
$ openssl genrsa -out https.key 2048
$ openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj
    /CN=www.kubia-example.com
```

Теперь, чтобы помочь лучше продемонстрировать несколько вещей о секретах, создайте дополнительный ничего не значащий файл под названием `foo`, и пусть он содержит строку `bar`. Вы поймете, почему вам нужно сделать это, через пару минут:

```
$ echo bar > foo
```

Теперь вы можете применить команду `kubectl create secret` для создания секрета из трех файлов:

```
$ kubectl create secret generic fortune-https --from-file=https.key
➥  --from-file=https.cert --from-file=foo
secret "fortune-https" created
```

Это не сильно отличается от создания словарей конфигурации. В этом случае вы создаете универсальный секрет под названием `fortune-https` и включаете в него две записи (https.key с содержимым файла https.key и аналогично для ключа/файла https.cert). Как вы узнали ранее, в место указания каждого файла по отдельности вы также можете включить весь каталог с помощью `--from-file=fortunehttps`.

> **ПРИМЕЧАНИЕ.** Вы создаете секрет общего типа, но вы могли бы также создать секрет `tls` с помощью команды `kubectl create secret tls`, как в главе 5. Правда, в результате будет создан секрет с разными именами записей.

7.5.4 Сравнение словарей конфигурации и секретов

Секреты и словари конфигурации имеют довольно большую разницу. Это то, что заставило разработчиков Kubernetes создать словари конфигурации после того, как Kubernetes уже некоторое время поддерживал секреты. В следующем ниже листинге показан YAML созданного вами секрета.

Листинг 7.21. Определение секрета на YAML

```
$ kubectl get secret fortune-https -o yaml
apiVersion: v1
data:
  foo: YmFyCg==
  https.cert: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCekNDQ...
  https.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcE...
kind: Secret
...
```

Теперь сравните это с YAML созданного ранее словаря конфигурации, который показан в следующем ниже листинге.

Листинг 7.22. Определение YAML в словаре ConfigMap

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  my-nginx-config.conf: |
    server {
      ...
    }
  sleep-interval: |
    25
kind: ConfigMap
...
```

Заметили разницу? Содержимое записей секрета отображается в виде строк в кодировке Base64, в то время как записи словаря конфигурации отображаются в виде открытого текста. Это первоначально делало работу с секретами в манифестах на YAML и JSON немного более болезненной, потому что вам приходилось кодировать и декодировать их при настройке и чтении записей.

Использование секретов для двоичных данных

Причина использования кодировки Base64 проста. Записи секрета могут содержать двоичные значения, а не только обычный текст. Кодировка Base64 позволяет включать двоичные данные в YAML или JSON, которые являются текстовыми форматами.

> **СОВЕТ.** Вы можете использовать секреты даже для нечувствительных двоичных данных, но помните, что максимальный размер секрета ограничен 1 Мб.

Знакомство с полем stringData

Поскольку не все чувствительные данные находятся в двоичной форме, Kubernetes также позволяет задавать значения секрета через поле `stringData`. В следующем ниже листинге показано, как оно используется.

Листинг 7.23. Добавление записей обычного текста в секрет с помощью поля stringData

```
kind: Secret
apiVersion: v1
stringData:              <-- Поле stringData может использоваться
                             для недвоичных данных секрета
  foo: plain text        <-- Смотрите, "plain-text" не закодирован в Base64
data:
  https.cert: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCekNDQ...
  https.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcE...
```

Поле `stringData` предназначено только для записи (еще раз: только для записи, а не чтения). Его можно использовать лишь для установки значений. При извлечении YAML секрета с помощью команды `kubectl get -o yaml` поле `stringData` не будет показано. Вместо этого все записи, указанные в поле `stringData` (например, запись `foo` в предыдущем примере), будут показаны в разделе `data` и будут закодированы в формате Base64, как и все другие записи.

Чтение записи секрета в модуле

Когда вы предоставляете контейнеру доступ к секрету через том `secret`, значение записи секрета декодируется и записывается в файл в его фактическом виде (независимо от того, является ли он обычным текстом или двоичным). То же самое верно и при обеспечении доступа к записи секрета через переменную среды. В обоих случаях приложению не нужно его декодировать, а просто прочитать содержимое файла или найти значение переменной среды и использовать его напрямую.

7.5.5 Использование секрета в модуле

Имея секрет `fortune-https`, содержащий файл и сертификата, и ключа для их использования, вам нужно всего лишь настроить веб-сервер Nginx.

Изменение словаря конфигурации fortune-config для активации https

Для этого необходимо снова изменить файл конфигурации путем редактирования словаря конфигурации:

```
$ kubectl edit configmap fortune-config
```

После открытия текстового редактора измените фрагмент, который определяет содержимое записи `my-nginxconfig.conf`, чтобы это выглядело как в следующем ниже листинге.

Листинг 7.24. Изменение данных словаря ConfigMap fortune-config

```
...
data:
  my-nginx-config.conf: |
```

```
    server {
      listen              80;
      listen              443 ssl;
      server_name         www.kubia-example.com;
      ssl_certificate     certs/https.cert;          Пути расположены
      ssl_certificate_key certs/https.key;           относительны /etc/nginx
      ssl_protocols TLSv1  TLSv1.1 TLSv1.2;
      ssl_ciphers         HIGH:!aNULL:!MD5;

      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }
    }
  sleep-interval: |
...
```

Это настраивает сервер на чтение файлов сертификата и ключей из /etc/nginx/certs, поэтому вам нужно будет там смонтировать том `secret`.

Монтирование секрета fortune-https в модуле

Затем вы создадите новый модуль fortune-https и смонтируете том `secret`, содержащий сертификат и ключ, в правильное месторасположение в контейнере `web-server`, как показано в следующем ниже листинге.

Листинг 7.25. Определение YAML модуля fortune-https: pod: fortune-pod-https.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-https
spec:
  containers:
  - image: luksa/fortune:env
    name: html-generator
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: fortune-config
          key: sleep-interval
    volumeMounts:
    - name: html
      mountPath: /var/htdocs
  - image: nginx:alpine
    name: web-server
    volumeMounts:
```

```
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
    - name: certs
      mountPath: /etc/nginx/certs/
      readOnly: true
    ports:
    - containerPort: 80
    - containerPort: 443
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    configMap:
      name: fortune-config
      items:
      - key: my-nginx-config.conf
        path: https.conf
  - name: certs
    secret:
    secretName: fortune-https
```

Вы сконфигурировали Nginx для чтения файла сертификата и ключа из /etc/nginx/certs, поэтому вам нужно смонтировать там том secret; вы определяете том secret здесь, ссылаясь на секрет fortune-https

Вы определяете том secret здесь, ссылаясь на секрет fortune-https

В этом описании модуля многое что происходит, поэтому давайте его визуализируем. На рис. 7.12 показаны компоненты, определенные в YAML. Секрет `default-token`, том и подключение тома, которые не являются частью YAML, но добавляются в модуль автоматически, на рисунке не показаны.

> **ПРИМЕЧАНИЕ.** Как и тома configMap, благодаря свойству `defaultMode` тома `secret` также поддерживают определение разрешений для файлов, доступных в томе.

Проверка использования сервером Nginx сертификата и ключа из секрета

После того как модуль запущен, вы можете увидеть, обслуживает ли он трафик HTTPS, открыв туннель переадресации портов к порту 443 модуля и используя его для отправки запроса на сервер с помощью `curl`:

```
$ kubectl port-forward fortune-https 8443:443 &
Forwarding from 127.0.0.1:8443 -> 443
Forwarding from [::1]:8443 -> 443
$ curl https://localhost:8443 -k
```

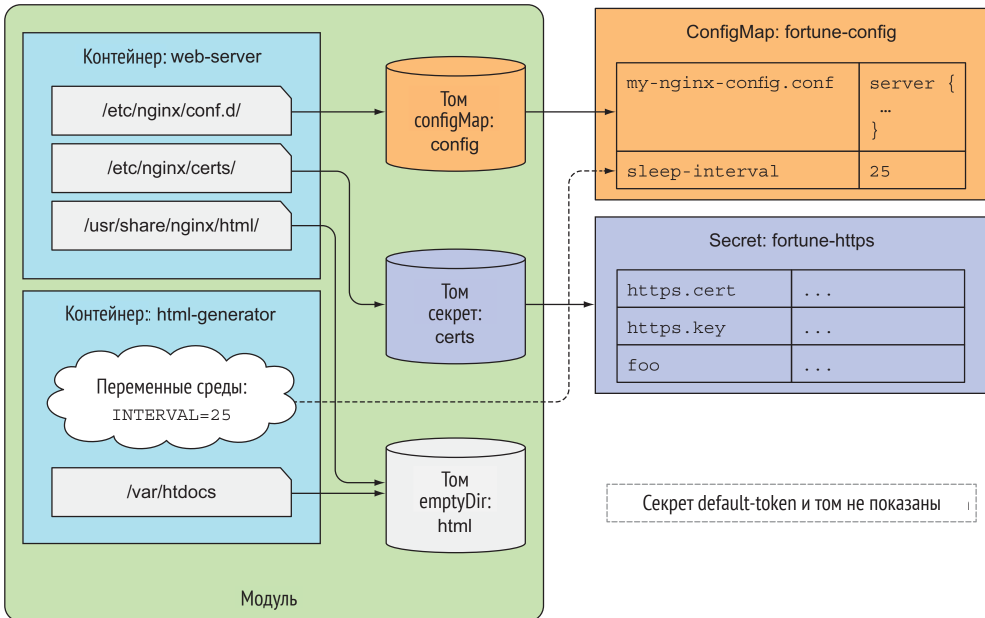


Рис. 7.12. Объединение томов configMap и секрета для выполнения вашего модуля fortune-https

Если вы правильно настроили сервер, вы должны получить отклик. Вы можете проверить сертификат сервера, чтобы увидеть, совпадает ли он с тем, который вы сгенерировали ранее. Это также можно сделать с помощью `curl`, включив ведение подробного журнала с помощью параметра `-v`, как показано в следующем ниже листинге.

Листинг 7.26. Вывод сертификата сервера, отправленного веб-сервером Nginx

```
$ curl https://localhost:8443 -k -v
* About to connect() to localhost port 8443 (#0)
* Trying ::1...
* Connected to localhost (::1) port 8443 (#0)
* Initializing NSS with certpath: sql:/etc/pki/nssdb
* skipping SSL peer certificate verification
* SSL connection using TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
* Server certificate:
*   subject: CN=www.kubia-example.com
*   start date: aug 16 18:43:13 2016 GMT
*   expire date: aug 14 18:43:13 2026 GMT
*   common name: www.kubia-example.com
*   issuer: CN=www.kubia-example.com
```

Сертификат совпадает с тем, который вы создали и сохранили в секрете

Тома secret хранятся в памяти

Вы успешно доставили свой сертификат и закрытый ключ в контейнер, смонтировав том `secret` в дереве каталогов /etc/nginx/certs. Для секретных файлов том `secret` использует файловую систему прямо в памяти (tmpfs). Вы можете увидеть это, если выведете результаты монтирования в контейнере:

```
$ kubectl exec fortune-https -c web-server -- mount | grep certs
tmpfs on /etc/nginx/certs type tmpfs (ro,relatime)
```

Поскольку используется tmpfs, чувствительные данные, хранящиеся в секрете, никогда не записываются на диск, где они могут быть поставлены под угрозу.

Предоставление доступа к записям секрета через переменные среды

Вместо того чтобы использовать том, вы могли бы также предоставить отдельные записи из секрета как переменные среды, как вы сделали с записью `sleep-interval` из словаря конфигурации. Например, если вы хотите предоставить ключ `foo` из своего секрета как переменную среды `FOO_SECRET`, вы должны добавить в определение контейнера фрагмент из следующего ниже листинга.

Листинг 7.27. Предоставление доступа к записи секрета как переменной среды

```
env:
- name: FOO_SECRET
  valueFrom:              # Переменная должна быть задана из записи секрета
    secretKeyRef:
      name: fortune-https # Имя секрета с ключом
      key: foo            # Предоставляемый ключ секрета
```

Это почти так же, как при установке переменной среды `INTERVAL`, за исключением того, что на этот раз вы ссылаетесь на секрет, используя `secretKeyRef` вместо `configMapKeyRef`, который применяется для ссылки на словарь конфигурации.

Несмотря на то что Kubernetes позволяет вам предоставлять доступ к секретам через переменные среды, применение этого функционального средства может оказаться не лучшей идеей. Приложения обычно сбрасывают переменные среды в отчеты об ошибках или даже записывают их в журнал приложений при запуске, что может непреднамеренно открыть к ним доступ. Кроме того, дочерние процессы наследуют все переменные среды родительского процесса, поэтому, если приложение запускает сторонний двоичный файл, невозможно узнать, что происходит с вашими секретными данными.

СОВЕТ. Подумайте хорошенько, прежде чем использовать переменные среды для передачи секретов в контейнер, поскольку они могут быть непреднамеренно раскрыты. Для того чтобы быть в безопасности, всегда используйте для предоставления доступа к секретам тома `secret`.

7.5.6 Секреты для выгрузки образов

Вы научились передавать секреты в свои приложения и использовать содержащиеся в них данные. Но иногда система Kubernetes сама требует, чтобы вы передавали ей учетные данные, например когда вы хотите использовать образы из приватного хранилища образов контейнеров. Это также делается посредством секретов.

До сих пор все ваши образы контейнеров хранились в общедоступных хранилищах образов, которые не требуют специальных учетных данных для извлечения из них образов. Но многие организации не хотят, чтобы их образы были доступны для всех, и поэтому они используют приватные хранилища образов. При развертывании модуля, образы контейнеров которого находятся в приватном реестре, Kubernetes требуется знать учетные данные, необходимые для извлечения образа. Давайте посмотрим, как это сделать.

Использование приватного репозитория в Docker Hub

Docker Hub, помимо общедоступных репозиториев образов, также позволяет создавать приватные репозитории. Вы можете отметить репозиторий как приватный, войдя в http://hub.docker.com с помощью вашего веб-браузера, найдя репозиторий и проставив галочку.

Для запуска модуля, использующего образ из приватного репозитория, необходимо выполнить два действия:

- создать секрет, содержащий учетные данные для реестра Docker;
- указать этот секрет в поле `imagePullSecrets` манифеста модуля.

Создание секрета для проверки подлинности в реестре Docker

Создание секрета, содержащего учетные данные для проверки подлинности в реестре Docker, не отличается от создания универсального секрета, созданного в разделе 7.5.3. Вы используете ту же команду `kubectl create secret`, но с другим типом и параметрами:

```
$ kubectl create secret docker-registry mydockerhubsecret \
--docker-username=myusername --docker-password=mypassword \
--docker-email=my.email@provider.com
```

Вместо того чтобы создавать универсальный секрет `generic`, вы создаете секрет `docker-registry` под названием `mydockerhubsecret`. Вы указываете имя пользователя, пароль и адрес электронной почты реестра Docker Hub. Если вы проверите содержимое вновь созданного секрета с помощью команды `kubectl describe`, то увидите, что он содержит единственную запись под на-

званием `.dockercfg`. Она эквивалентна файлу .dockercfg в вашем домашнем каталоге, который создается платформой Docker при выполнении команды `docker login`.

Использование секрета docker-registry в определении модуля

Для того чтобы система Kubernetes использовала секрет при извлечении образов из приватного хранилища реестра Docker Hub, вам нужно только указать имя секрета в спецификации модуля, как показано в следующем ниже листинге.

Листинг 7.28. Определение модуля с использованием секрета для извлечения образа: pod-with-private-image.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: private-pod
spec:
  imagePullSecrets:
  - name: mydockerhubsecret
  containers:
  - image: username/private:tag
    name: main
```

Это позволяет извлекать образы из приватного реестра образов

В определении модуля в приведенном выше листинге вы указываете секрет `mydockerhubsecret` как один из секретов `imagePullSecret`. Я предлагаю вам попробовать это самостоятельно, потому что вы, весьма вероятно, скоро будете иметь дело с приватными образами контейнеров.

Отсутствие необходимости указывать секреты для выгрузки образов на каждом модуле

Учитывая, что люди в своих системах обычно запускают много разных модулей, это заставляет вас задуматься над вопросом, нужно ли вам добавлять одни и те же секреты выгрузки образов в каждый модуль. К счастью, не нужно. В главе 12 вы узнаете, как можно автоматически добавлять секреты извлечения образов во все ваши модули, если добавлять секреты в ServiceAccount.

7.6 Резюме

В данном разделе мы подведем итог этой главы, посвященной тому, как передать конфигурационные данные в контейнеры. Вы научились:

- переопределять команду по умолчанию, заданную в образе контейнера в определении модуля;
- передавать аргументы командной строки главному процессу контейнера;

- устанавливать переменные среды для контейнера;
- отделять конфигурацию от спецификации модуля и помещать ее в словарь конфигурации ConfigMap;
- хранить конфиденциальные данные в секрете Secret и надежно доставлять их в контейнеры;
- создавать секрет `docker-registry` и использовать его для извлечения образов из приватного реестра образов.

В следующей главе вы узнаете, как передавать метаданные модулей и контейнеров в запущенные внутри них приложения. Вы также увидите, как секрет с токеном по умолчанию `default-token`, о котором мы узнали в этой главе, используется для обмена с сервером API изнутри модуля.

Глава 8

Доступ к метаданным модуля и другим ресурсам из приложений

Эта глава посвящена:

- использованию Downward API для передачи информации в контейнеры;
- изучению REST API Kubernetes;
- передаче аутентификации и верификации сервера в команду `kubectl proxy`;
- доступу к серверу API из контейнера;
- общим сведениям о шаблоне контейнера-посредника (Ambassador);
- использованию клиентских библиотек Kubernetes.

Приложениям часто требуется информация о среде, в которой они работают, включая сведения о себе и других компонентах кластера. Вы уже видели, как система Kubernetes обеспечивает обнаружение служб с помощью переменных среды или DNS, но как насчет другой информации? В этой главе вы увидите, как определенные метаданные модулей и контейнеров могут передаваться в контейнер и как легко приложение, запущенное в контейнере, может связываться с сервером API Kubernetes, чтобы получать информацию о ресурсах, развернутых в кластере, и даже о том, как создавать или изменять эти ресурсы.

8.1 Передача метаданных через нисходящий API

В предыдущей главе вы увидели, как передавать конфигурационные данные в приложения через переменные среды или через тома `configMap` и `secret`. Это

хорошо работает для данных, которые вы устанавливаете сами и которые известны до того, как модуль будет назначен узлу и там запущен. Но как насчет данных, которые не известны до этого момента, – таких как IP-адрес модуля, имя узла или даже собственное имя модуля (когда имя генерируется; например, когда модуль создается объектом ReplicaSet или подобным контроллером)? А как насчет данных, которые уже указаны в другом месте, таких как метки и аннотации модуля? Вы же не хотите повторять одну и ту же информацию в нескольких местах.

Обе эти проблемы решаются Kubernetes Downward API. Он позволяет передавать метаданные о модуле и его среде через переменные среды или файлы (в томе `downwardAPI`). Не ошибайтесь в названии. Downward API не похож на REST, к которому должно обратиться приложение, чтобы получить данные. Он представляет собой способ заполнения переменных среды или файлов значениями из спецификации или состояния модуля, как показано на рис. 8.1.

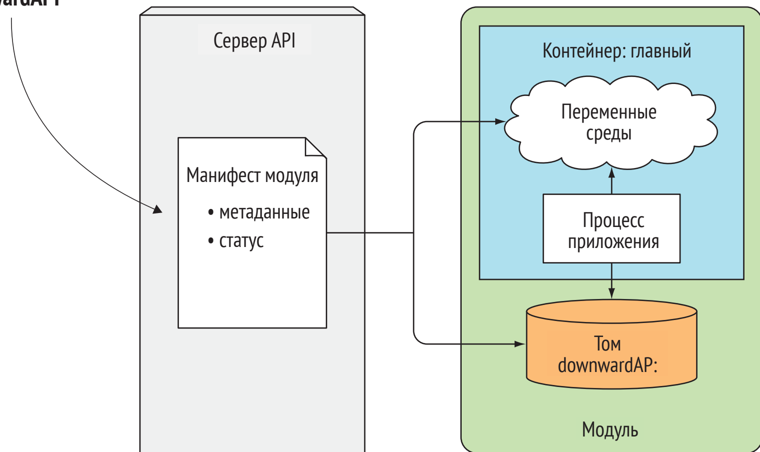


Рис. 8.1. Downward API предоставляет доступ к метаданным модуля через переменные среды или файлы

8.1.1 Доступные для использования метаданные

Нисходящий API позволяет предоставлять собственные метаданные модуля процессам, запущенным внутри этого модуля. В настоящее время он дает возможность передавать в контейнеры следующую информацию:

- имя модуля;
- IP-адрес модуля;
- пространство имен, к которому принадлежит модуль;
- имя узла, на котором модуль работает;
- имя учетной записи службы, под которой модуль работает;
- запросы ЦП и памяти для каждого контейнера;

- ограничения ЦП и памяти для каждого контейнера;
- метки модуля;
- аннотации модуля.

Большинство элементов в списке не должны требовать дальнейшего объяснения, за исключением, возможно, учетной записи службы и запросов и ограничений ЦП/памяти, с которыми мы еще не знакомились. Мы подробно рассмотрим учетные записи служб в главе 12. Сейчас же вам нужно знать только, что учетная запись службы – это учетная запись, которую модуль проверяет при обмене с сервером API. Запросы и ограничения ЦП и памяти описаны в главе 14. Это объем процессора и памяти, гарантированный контейнеру, и максимальный объем, который он может получить.

Большинство элементов в этом списке может передаваться в контейнеры либо через переменные среды, либо через том `downwardAPI`, но метки и аннотации могут предоставляться только через том. Часть данных может быть получена другими способами (например, непосредственно из операционной системы), однако Downward API предоставляет более простую альтернативу.

Давайте посмотрим на пример передачи метаданных в ваш контейнеризированный процесс.

8.1.2 Предоставление доступа к метаданным через переменные среды

Прежде всего давайте посмотрим, каким образом можно передавать метаданные модуля и контейнера в контейнер через переменные среды. Вы создадите простой одноконтейнерный модуль из манифеста следующего ниже листинга.

Листинг 8.1. Downward API, использован для передачи переменных среды: downward-api-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: downward
spec:
  containers:
  - name: main
    image: busybox
    command: ["sleep", "9999999"]
    resources:
      requests:
        cpu: 15m
        memory: 100Ki
      limits:
        cpu: 100m
```

```
        memory: 4Mi
    env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: SERVICE_ACCOUNT
      valueFrom:
        fieldRef:
          fieldPath: spec.serviceAccountName
    - name: CONTAINER_CPU_REQUEST_MILLICORES
      valueFrom:
        resourceFieldRef:
          resource: requests.cpu
          divisor: 1m
    - name: CONTAINER_MEMORY_LIMIT_KIBIBYTES
      valueFrom:
        resourceFieldRef:
          resource: limits.memory
          divisor: 1Ki
```

Вместо указания абсолютного значения вы ссылаетесь на поле metadata.name из манифеста модуля

Ссылка на запросы и лимиты на ЦП и память контейнера реализуется с помощью поля resourceFieldRef, а не поля fieldRef

Для ресурсных полей вы определяете делитель divisor, чтобы получить значение в нужной вам единице измерения

При запуске процесса он может просмотреть все переменные среды, определенные в спецификации модуля. На рис. 8.2 показаны переменные среды и источники их значений. Название модуля, его IP и пространство имен, которые будут представлены через переменные среды, – соответственно `POD_NAME`, `POD_IP` и `POD_NAMESPACE`. Имя узла, на котором выполняется контейнер, будет предоставлено через переменную `NODE_NAME`. Имя учетной записи службы становится доступным через переменную среды `SERVICE_ACCOUNT`. Вы также создаете две переменные среды, которые будут содержать объем ЦП, запрошенный для этого контейнера, и максимальный объем памяти, который контейнер может использовать.

Для переменных среды, предоставляющих доступ к ресурсным лимитам или запросам, указывается делитель. Фактическое значение лимита или запроса будет разделено на делитель, и результат будет предоставлен через перемен-

ную среды. В предыдущем примере вы устанавливаете делитель для запросов ЦП, равный `1m` (одно миллиядро, или *одна* тысячная ядра ЦП). Поскольку для запроса ЦП задано значение `15m`, переменной среды `CONTAINER_CPU_REQUEST_MILLICORES` будет присвоено значение 15. Кроме того, вы можете установить лимит памяти в `4Mi` (4 мебибайта) и делитель в `1Ki` (1 кибибайт), поэтому переменная среды `CONTAINER_MEMORY_LIMIT_KIBIBYTES` будет равняться `4096`.

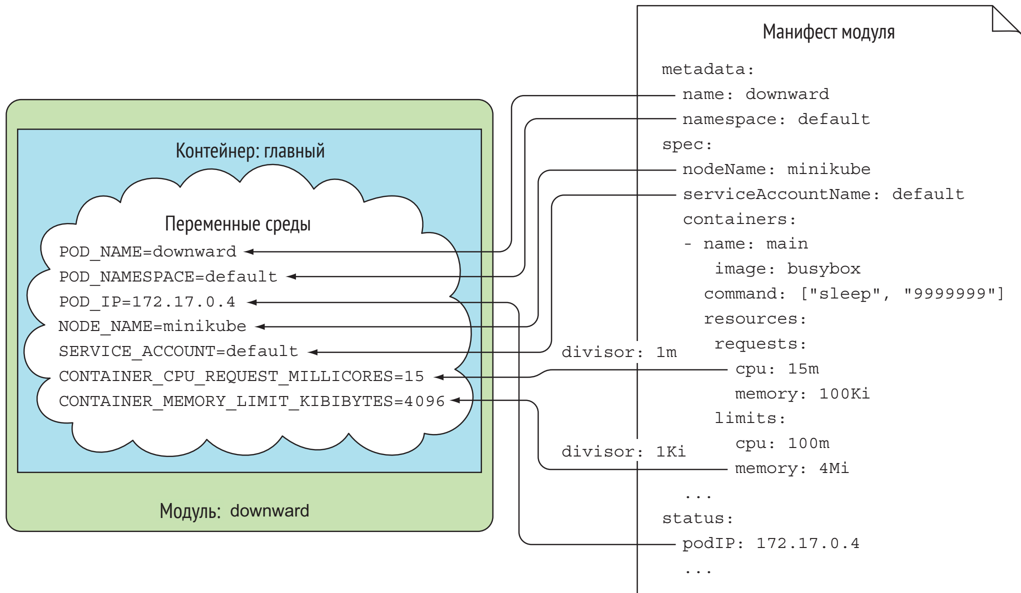


Рис. 8.2. Метаданные и атрибуты Pod могут быть представлены pod через переменные среды

Делитель для лимитов и запросов на ЦП может быть либо `1`, что означает одно целое ядро, либо `1m`, что составляет одно миллиядро. Делитель для лимитов/запросов на память может быть `1` (байт), `1K` (килобайт) или `1Ki` (кибибайт), `1M` (мегабайт) или `1Mi` (мебибайт) и т. д.

После создания модуля можно применить команду `kubectl exec` для просмотра всех этих переменных среды внутри вашего контейнера, как показано в следующем ниже листинге.

Листинг 8.2. Переменные среды в нисходящем модуле

```
$ kubectl exec downward env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=downward
CONTAINER_MEMORY_LIMIT_KIBIBYTES=4096
POD_NAME=downward
POD_NAMESPACE=default
POD_IP=10.0.0.10
NODE_NAME=gke-kubia-default-pool-32a2cac8-sgl7
SERVICE_ACCOUNT=default
```

```
CONTAINER_CPU_REQUEST_MILLICORES=15
KUBERNETES_SERVICE_HOST=10.3.240.1
KUBERNETES_SERVICE_PORT=443
...
```

Все процессы, запущенные внутри контейнера, могут читать эти переменные и использовать их по мере необходимости.

8.1.3 Передача метаданных через файлы в том downwardAPI

Если вы предпочитаете предоставлять доступ к метаданным через файлы, а не через переменные среды, то вы можете определить том `downwardAPI` и смонтировать его в контейнер. Вы должны использовать том `downwardAPI` для предоставления доступа к меткам модуля или его аннотациям, поскольку ни к одному из них не может быть представлен доступ через переменные среды. Позже мы обсудим, почему.

Как и в случае переменных среды, вам нужно явно указать каждое поле метаданных, если вы хотите, чтобы оно было доступно процессу. Давайте посмотрим, как изменить предыдущий пример, чтобы вместо переменных среды использовать том, как показано в следующем ниже листинге.

Листинг 8.3. Модуль с томом downwardAPI: downward-api-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: downward
  labels:
    foo: bar
  annotations:
    key1: value1
    key2: |
      multi
      line
      value
spec:
  containers:
  - name: main
    image: busybox
    command: ["sleep", "9999999"]
    resources:
      requests:
        cpu: 15m
        memory: 100Ki
      limits:
```

Эти метки и аннотации будут доступны через том downwardAP. Вы монтируете том downwardAPI в /etc/вниз

```
      cpu: 100m
      memory: 4Mi
    volumeMounts:
    - name: downward
      mountPath: /etc/downward
  volumes:
  - name: downward
    downwardAPI:
      items:
      - path: "podName"
        fieldRef:
          fieldPath: metadata.name
      - path: "podNamespace"
        fieldRef:
          fieldPath: metadata.namespace
      - path: "labels"
        fieldRef:
          fieldPath: metadata.labels
      - path: "annotations"
        fieldRef:
          fieldPath: metadata.annotations
      - path: "containerCpuRequestMilliCores"
        resourceFieldRef:
          containerName: main
          resource: requests.cpu
          divisor: 1m
      - path: "containerMemoryLimitByte"
        resourceFieldRef:
          containerName: main
          resource: limits.memory
          divisor: 1
```

Эти метки и аннотации будут доступны через том downwardAP. Вы монтируете том downwardAPI в /etc/вниз

Вы определяете том downwardAPI с именем downward

Имя модуля (из поля metadata.name в манифесте) будет записано в файл podName

Метки модуля будут записаны в файл /etc/downward/labels

Аннотации модуля будут записаны в файл /etc/downward /annotations

Вместо передачи метаданных через переменные среды вы определяете том с именем `downward` и монтируете его в контейнере в /etc/downward. Файлы, содержащиеся в этом томе, будут сконфигурированы под атрибутом `downwardAPI.items` в спецификации тома.

Каждый элемент указывает путь `path` (имя файла), куда должны быть записаны метаданные, и ссылается на поле уровня модуля либо на ресурсное поле контейнера, значение которого вы хотите сохранить в файле (см. рис. 8.3).

Удалите предыдущую запись и создайте новую из манифеста в приведенном выше листинге. Затем просмотрите содержимое каталога смонтированного тома `downwardAPI`. Вы смонтировали том в /etc/downward/, поэтому выведите список файлов в этом месторасположении, как показано в следующем ниже листинге.

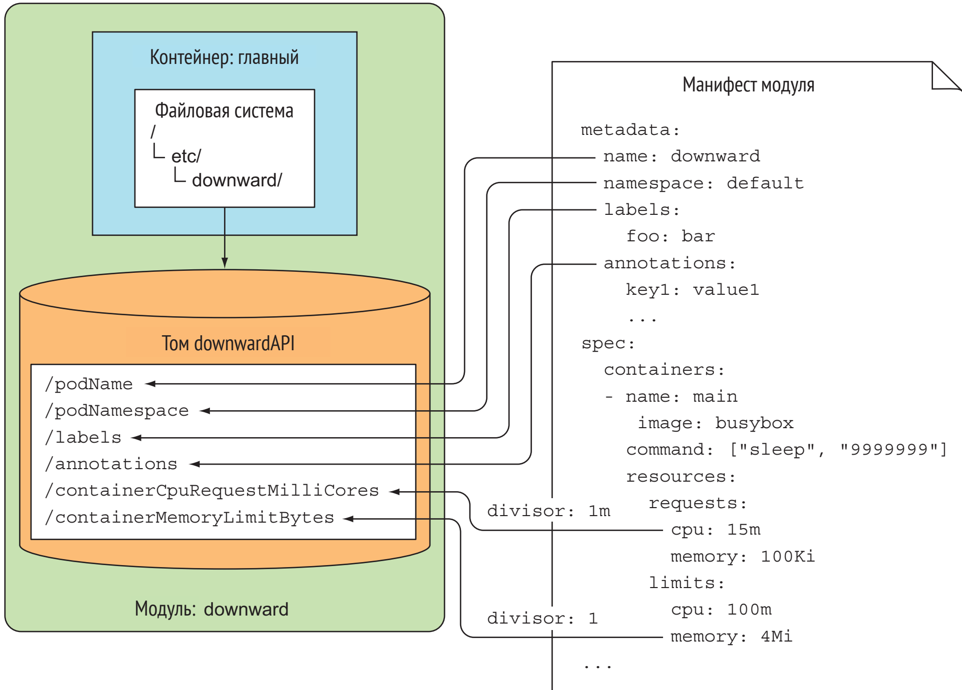


Рис. 8.3. Использование тома downwardAPI для передачи метаданных в контейнер

Листинг 8.4. Файлы в нисходящем томе API

```
$ kubectl exec downward ls -lL /etc/downward
-rw-r--r-- 1  root  root  134  May 25 10:23 annotations
-rw-r--r-- 1  root  root    2  May 25 10:23 containerCpuRequestMilliCores
-rw-r--r-- 1  root  root    7  May 25 10:23 containerMemoryLimitBytes
-rw-r--r-- 1  root  root    9  May 25 10:23 labels
-rw-r--r-- 1  root  root    8  May 25 10:23 podName
-rw-r--r-- 1  root  root    7  May 25 10:23 podNamespace
```

ПРИМЕЧАНИЕ. Как в случае с томами configMap и `secret`, вы можете изменить разрешения файлов через свойство `defaultMode` тома `downwardAPI` в секции spec модуля.

Каждый файл соответствует элементу в определении тома. Содержимое файлов, которые соответствуют тем же полям метаданных, что и в предыдущем примере, совпадает со значениями переменных среды, которые вы использовали ранее, поэтому мы не будем их здесь показывать. Но поскольку раньше вы не могли предоставлять доступ к меткам и примечаниям через переменные среды, проинспектируйте следующий ниже листинг в отношении содержимого двух файлов, в которых вы предоставили к ним доступ.

Листинг 8.5. Вывод меток и аннотаций в томе downwardAPI

```
$ kubectl exec downward cat /etc/downward/labels
foo="bar"

$ kubectl exec downward cat /etc/downward/annotations
key1="value1"
key2="multi\nline\nvalue\n"
kubernetes.io/config.seen="2016-11-28T14:27:45.664924282Z"
kubernetes.io/config.source="api"
```

Как вы можете видеть, каждая метка/аннотация записывается в формате `ключ=значение` на отдельной строке. Многострочные значения записываются в одну строку с символами новой строки, обозначаемыми с помощью `\n`.

Обновление меток и аннотаций

Вы, возможно, помните, что метки и аннотации могут быть изменены во время работы модуля. Как и следовало ожидать, при их изменении Kubernetes обновляет содержащие их файлы, позволяя модулю всегда видеть актуальные данные. Это также объясняет, почему метки и аннотации не могут быть представлены через переменные среды. Если доступ к меткам или аннотациям модуля был предоставлен через переменные среды, то с учетом того, что значения переменных среды впоследствии не могут обновляться, нет никакого способа обеспечить доступ к новым значениям после их изменения.

Ссылка на метаданные уровня контейнера в спецификации тома

Прежде чем мы закончим этот раздел, мы должны отметить один аспект. При предоставлении метаданных уровня контейнера, таких как ресурсные лимиты или запросы контейнера (выполняемые с помощью `resourceFieldRef`), вам нужно указать имя контейнера, на чье поле ресурса вы ссылаетесь. Это показано в следующем ниже листинге.

Листинг 8.6. Ссылка на метаданные уровня контейнера в томе downwardAPI

```
spec:
  volumes:
  - name: downward
    downwardAPI:
      items:
      - path: "containerCpuRequestMilliCores"
        resourceFieldRef:
          containerName: main        <- Должно быть указано
          resource: requests.cpu        имя контейнера
          divisor: 1m
```

Причина этого становится очевидной, если учесть, что тома определяются на уровне модуля, а не на уровне контейнера. При ссылке на ресурсное поле контейнера в спецификации тома необходимо явно указать имя контейнера, на который вы ссылаетесь. Это истинно даже для одноконтейнерных модулей.

Использование томов для предоставления доступа к ресурсным запросам и/или лимитам контейнера несколько сложнее, чем использование переменных среды, но их преимущество заключается в том, что при необходимости можно передавать ресурсные поля одного контейнера другому контейнеру (при этом оба контейнера должны находиться в одном модуле). В случае с переменными среды контейнеру можно передавать только собственные ресурсные лимиты и запросы.

Когда использовать Downward API

Как вы видели, использовать Downward API не сложно. Он позволяет держать приложение платформенно-независимым от Kubernetes. Это особенно полезно при работе с существующим приложением, которое ожидает наличия определенных данных в переменных среды. Downward API позволяет предоставлять данные приложению без необходимости переписывать приложение или оборачивать его в шелл-скрипт, который собирает данные и затем предоставляет к ним доступ через переменные среды.

Но метаданные, доступные через Downward API, довольно ограничены. Если вам нужно больше, то следует получать их непосредственно с сервера API Kubernetes. Далее вы узнаете, как это сделать.

8.2 Обмен с сервером API Kubernetes

Мы видели, что Downward API обеспечивает простой способ передачи определенных метаданных модуля и контейнера процессу, работающему внутри них. Он предоставляет доступ только к собственным метаданным модуля и подмножеству всех данных модуля. Но иногда приложение должно знать о других модулях больше и даже о других ресурсах, определенных в кластере. В этих случаях Downward API не помогает.

Как вы видели в книге, данные о службах и модулях можно получить, обратившись к переменным среды, связанным со службами, либо через DNS. Но когда приложению нужны данные о других ресурсах или когда ему требуется доступ к самым последним возможным данным, ему нужно связываться с сервером API напрямую (как показано на рис. 8.4).

Прежде чем вы увидите, как приложения в модулях могут взаимодействовать с сервером API Kubernetes, давайте сначала рассмотрим конечные точки REST сервера со стороны вашей локальной машины, чтобы вы могли увидеть, как выглядит обмен с сервером API.

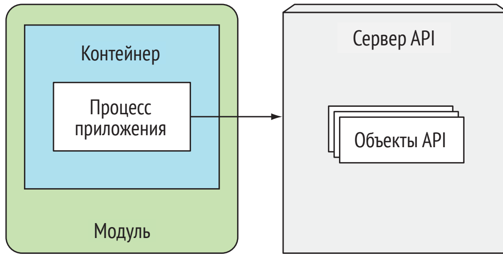


Рис. 8.4. Обмен с сервером API изнутри модуля для получения данных о других объектах API

8.2.1 Исследование REST API Kubernetes

Вы познакомились с различными типами ресурсов Kubernetes. Но если вы планируете разрабатывать приложения, которые взаимодействуют с API Kubernetes, то сначала вы захотите познакомиться с его API.

Для этого вы можете попробовать выйти на сервер API непосредственно. Вы можете получить его URL, выполнив команду `kubectl cluster-info`:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
```

Поскольку этот сервер использует протокол HTTPS и требует аутентификации, общаться напрямую с ним не просто. Вы можете попробовать получить к нему доступ с помощью утилиты `curl` и использовать ее параметр `--insecure` (или `-k`), чтобы пропустить проверку сертификата сервера, но это несильно вам поможет:

```
$ curl https://192.168.99.100:8443 -k
Unauthorized
```

К счастью, вместо того чтобы заниматься аутентификацией самостоятельно, вы можете обмениваться с сервером через прокси, выполнив команду `kubectl proxy`.

Доступ к серверу API через прокси kubectl

Команда `kubectl proxy` запускает прокси-сервер, который принимает HTTP-подключения на локальном компьютере и передает их на сервер API, обеспечивая при этом аутентификацию, поэтому вам не нужно передавать токен аутентификации в каждом запросе. Это также гарантирует, что вы обмениваетесь с фактическим сервером API, а не с незаконным посредником (проверяющим сертификат сервера по каждому запросу).

Запуск прокси-сервера тривиален. Вам нужно всего лишь выполнить следующую ниже команду:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Вам не требуется передавать какие-либо другие аргументы, потому что `kubectl` уже знает все, что ему нужно (URL-адрес сервера API, токен авториза-

ции и т. д.). Как только он запускается, прокси начинает принимать подключения на локальном порту 8001. Посмотрим, сработает ли это:

```
$ curl localhost:8001
{
  "paths": [
    "/api",
    "/api/v1",
...
```

Вуаля! Вы отправили запрос к прокси, он направил запрос на сервер API, и прокси вернул все, что вернул сервер. Теперь давайте начнем исследовать.

Исследование API Kubernetes через команду kubectl proxy

Вы можете продолжать использовать утилиту `curl` либо можете открыть свой веб-браузер и направить его на http://localhost:8001. Давайте внимательнее рассмотрим, что сервер API возвращает, когда вы выходите на его базовый URL. Сервер откликается списком путей, как показано в следующем ниже листинге.

Листинг 8.7. Вывод списка конечных точек REST сервера API: http://localhost:8001

```
$ curl http://localhost:8001
{
  "paths": [
    "/api",
    "/api/v1",                  <- Большинство типов ресурсов
                                   можно найти здесь
    "/apis",
    "/apis/apps",
    "/apis/apps/v1beta1",
    ...
    "/apis/batch",              <- Группа API batch и две
    "/apis/batch/v1",              ее версии
    "/apis/batch/v2alpha1",
    ...
```

Эти пути соответствуют группам и версиям API, указанным в определениях ресурсов при создании таких ресурсов, как Pod, Service и т. д.

Вы можете распознать `batch/v1` в пути `/apis/batch/v1` как группы и версии API ресурсов Job, о которых вы узнали в главе 4. Точно так же `/api/v1` соответствует `apiVersion: v1`, на которую вы ссылаетесь в созданных вами общих ресурсах (модуль, служба, контроллеры репликации и т. д.). Наиболее распространенные типы ресурсов, которые были введены в самых ранних версиях Kubernetes, не принадлежат ни к какой конкретной группе, потому что в Kubernetes концепция групп API изначально даже не использовалась; она была введена позже.

ПРИМЕЧАНИЕ. Эти исходные типы ресурсов без группы API сейчас входят в состав ключевой группы API.

Изучение конечной точки REST группы API batch

Давайте рассмотрим API ресурса Job. Для начала вы взглянете на то, что стоит за путем `/apis/batch` (версию вы пока отложите в сторону), как показано в следующем ниже листинге.

Листинг 8.8. Вывод списка конечных точек в /apis/batch: http://localhost:8001/apis/batch

```
$ curl http://localhost:8001/apis/batch
{
  "kind": "APIGroup",
  "apiVersion": "v1",
  "name": "batch",
  "versions": [
    {
      "groupVersion": "batch/v1",
      "version": "v1"
    },
    {
      "groupVersion": "batch/v2alpha1",
      "version": "v2alpha1"
    }
  ],
  "preferredVersion": {
    "groupVersion": "batch/v1",
    "version": "v1"
  },
  "serverAddressByClientCIDRs": null
}
```

Группа API batch содержит две версии

Клиенты должны использовать версию v1 вместо v2alpha1

В ответе показано описание группы API batch, включая доступные версии и предпочтительную версию, которую должны использовать клиенты. Давайте продолжим и посмотрим, что стоит за путем `/apis/batch/v1`. Это показано в следующем ниже листинге.

Листинг 8.9. Типы ресурсов в batch/vl: http://localhost:8001/apis/batch/v1

```
$ curl http://localhost:8001/apis/batch/v1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "batch/v1",
  "resources": [
```

Это список ресурсов API в группе API batch/v1

Вот массив, содержащий все типы ресурсов в этой группе

```
    {
      "name": "jobs",
      "namespaced": true,
      "kind": "Job",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch"
      ]
    },
    {
      "name": "jobs/status",
      "namespaced": true,
      "kind": "Job",
        "verbs": [
        "get",
        "patch",
        "update"
      ]
    }
  ]
}
```

Описывает ресурс задания, то есть организованный в пространство имен

Вот команды, которые можно использовать с этим ресурсом (можно создавать задания, удалять отдельные задания или их коллекцию, а также их извлекать, просматривать и обновлять)

Ресурсы также имеют специальную конечную точку REST для изменения их состояния

Состояние может быть извлечено, исправлено или обновлено

Как видите, в группе API `batch/v1` сервер API возвращает список типов ресурсов и конечных точек REST. Одним из них является ресурс Job. Помимо имени `name` ресурса и связанного с ним вида `kind`, сервер API также содержит информацию о том, организован и ресурс в пространство имен или нет (`namespaced`), его краткое имя (если есть; у заданий его нет) и список команд, которые можно использовать с ресурсом.

Возвращаемый список описывает ресурсы REST, предоставляемые на сервере API. Строка `"name": "jobs"` указывает, что API содержит конечную точку `/apis/batch/v1/jobs`. В массиве глаголов `"verbs"` указано, что ресурсы Job можно извлекать, обновлять и удалять через эту конечную точку. Для определенных ресурсов также доступны дополнительные конечные точки API (например, путь `jobs/status`, который позволяет изменять только статус задания).

Список всех экземпляров заданий в кластере

Чтобы получить список заданий в кластере, выполните запрос GET к пути `/apis/batch/v1/jobs`, как показано в следующем ниже листинге.

Листинг 8.10. Список заданий: http://localhost:8001/apis/batch/v1/jobs

```
$ curl http://localhost:8001/apis/batch/v1/jobs
{
  "kind": "JobList",
  "apiVersion": "batch/v1",
  "metadata": {
    "selfLink": "/apis/batch/v1/jobs",
    "resourceVersion": "225162"
  },
  "items": [
    {
      "metadata": {
      "name": "my-job",
      "namespace": "default",
      ...
```

Возможно, в кластере не развернуты ресурсы заданий, поэтому массив элементов будет пустым. Можно попробовать развернуть задание в Chapter08/my-job.yaml и повторно выйти на конечную точку REST для получения того же результата, что и в листинге 8.10.

Получение определенного экземпляра задания по имени

Предыдущая конечная точка возвращала список всех заданий во всех пространствах имен. Чтобы вернуть только одно конкретное задание, необходимо в URL-адресе указать его имя и пространство имен. Чтобы получить задание, показанное в предыдущем списке (`name: my-job; namespace: default`), необходимо запросить следующий путь: `/apis/batch/v1/namespaces/default/jobs/my-job`, как показано в приведенном ниже листинге.

Листинг 8.11. Получение ресурса в определенном пространстве имен по имени

```
$ curl http://localhost:8001/apis/batch/v1/namespaces/default/jobs/my-job
{
  "kind": "Job",
  "apiVersion": "batch/v1",
  "metadata": {
    "name": "my-job",
    "namespace": "default",
    ...
```

Как вы можете видеть, обратно вы получаете полное определение ресурса задания `my-job` на JSON, точно так же, как и при выполнении команды:

```
$ kubectl get job my-job -o json
```

Вы убедились, что можете просматривать сервер API REST Kubernetes без использования каких-либо специальных инструментов, но, чтобы полностью

изучить API REST и взаимодействовать с ним, более оптимальный вариант описан в конце этой главы. На данный момент, исследуя его с помощью утилиты `curl`, как здесь, достаточно, чтобы вы поняли, каким образом работающее в модуле приложение обменивается с Kubernetes.

8.2.2 Обмен с сервером API изнутри модуля

Вы научились обмениваться с сервером API с локального компьютера, используя прокси-сервер `kubectl`. Теперь давайте посмотрим, как взаимодействовать с ним изнутри модуля, где у вас (обычно) нет `kubectl`. Поэтому, чтобы обменяться с сервером API изнутри модуля, вам нужно позаботиться о трех вещах:

- найти расположение сервера API;
- убедиться, что вы обмениваетесь с сервером API, а не с тем, что пытается им притворяться;
- аутентифицироваться на сервере; в противном случае он не позволит вам ничего увидеть или сделать.

Вы увидите, как это делается, в следующих трех разделах.

Запуск модуля для проверки связи с сервером API

Первое, что вам нужно, – это модуль, из которого можно взаимодействовать с сервером API. Вы запустите модуль, который ничего не делает (он выполняет команду sleep в своем единственном контейнере), а затем запустите оболочку в контейнере с помощью `kubectl exec`. Затем вы попытаетесь получить доступ к серверу API из этой оболочки с помощью утилиты `curl`.

Поэтому вам нужно использовать образ контейнера, содержащий двоичный файл `curl`. Если вы ищете такой образ, скажем, в Docker Hub, то вы найдете образ `tutum/curl`, поэтому используйте его (вы также можете использовать любой другой существующий образ, содержащий исполняемый файл `curl`, или можете создать свой собственный). Определение модуля показано в следующем ниже листинге.

Листинг 8.12. Модуль для опробирования обмена с сервером API: curl.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: curl
spec:
  containers:
  - name: main
    image: tutum/curl
    command: ["sleep", "9999999"]
```

Использование образа tutum/curl, потому что вам в контейнере нужна утилита curl

Вы выполняете команду sleep с большой задержкой, чтобы ваш контейнер продолжал работать

После создания модуля выполните команду `kubectl exec`, чтобы запустить оболочку bash внутри контейнера:

```
$ kubectl exec -it curl bash
root@curl:/#
```

Теперь вы готовы к взаимодействию с сервером API.

Поиск адреса сервера API

Прежде всего вам нужно найти IP и порт сервера API Kubernetes. Это легко, поскольку доступ к службе под названием `kubernetes` автоматически предоставляется в пространстве имен по умолчанию и настраивается для указания на сервер API. Вы, возможно, помните, что видели его всякий раз, когда вы выводили список служб с помощью команды `kubectl get svc`:

```
$ kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.0.0.1     <none>        443/TCP   46d
```

И вы помните из главы 5, что переменные среды сконфигурированы для каждой службы. IP-адрес и порт сервера API можно получить, просмотрев переменные `KUBERNETES_SERVICE_HOST` и `KUBERNETES_SERVICE_PORT` (внутри контейнера):

```
root@curl:/# env | grep KUBERNETES_SERVICE
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Вы, возможно, помните, что каждая служба – также DNS-запись, поэтому вам даже не нужно искать переменные среды, а просто нужно направить утилиту `curl` на https://kubernetes. По правде говоря, если вы не знаете, на каком порту доступна служба, то, для того чтобы получить фактический номер порта службы, вы также должны либо отыскать переменные среды, либо выполнить поиск записи SRV DNS.

Переменные среды, показанные ранее, говорят о том, что сервер API прослушивает порт 443, который по умолчанию является портом для HTTPS, поэтому попытайтесь выйти на сервер через HTTPS:

```
root@curl:/# curl https://kubernetes
curl: (60) SSL certificate problem: unable to get local issuer certificate
...
If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

Хотя самый простой способ обойти это – использовать предложенный параметр `-k` (и это то, что вы обычно используете во время экспериментирования с сервером API вручную), давайте посмотрим на более длинный (и пра-

вильный) путь. Вместо того чтобы слепо доверять, что сервер, к которому вы подключаетесь, является аутентичным сервером API, вы проверите его идентичность, проверив его сертификат.

> **СОВЕТ.** Никогда не пропускайте проверку сертификата сервера в реальном приложении. Это может сделать токен аутентификации вашего приложения доступным для злоумышленника, использующего атаку с применением технологии «man-in-the-middle».

Проверка удостоверения сервера

В предыдущей главе, обсуждая секреты, мы рассмотрели автоматически созданный секрет под названием `default-token-xyz`, который монтируется в каждый контейнер в /var/run/secrets/kubernetes.io/serviceaccount/. Давайте снова взглянем на содержимое этого секрета, выведя список файлов в этом каталоге:

```
root@curl:/#ls/var/run/secrets/kubernetes.io/serviceaccount/
ca.crt    namespace    token
```

Секрет имеет три записи (и, следовательно, три файла в томе `secret`). Сейчас мы сосредоточимся на файле ca.crt, содержащем сертификат центра сертификации (certificate authority, CA), используемый для подписания сертификата сервера API Kubernetes. Чтобы проверить, что вы обмениваетесь с сервером API, необходимо проверить, подписан сертификат сервера центром сертификации или нет. Утилита `curl` позволяет указать сертификат CA с помощью параметра `--cacert`, поэтому попробуйте снова выйти на сервер API:

```
root@curl:/# curl --cacert /var/run/secrets/kubernetes.io/serviceaccount
          ➥ /ca.crt https://kubernetes
Unauthorized
```

> **ПРИМЕЧАНИЕ.** Вы можете увидеть более подробное описание ошибки, чем просто «Unauthorized» (Не санкционировано).

Хорошо, вы добились прогресса. Утилита `curl` проверила удостоверение сервера, так как его сертификат был подписан доверенным центром сертификации. Как следует из отклика `Unauthorized`, вам все равно нужно позаботиться об аутентификации. Вы сделаете это чуть позже, но сначала давайте посмотрим, как упростить жизнь, установив переменную среды `CURL_CA_BUNDLE`, чтобы вам не нужно было указывать параметр `--cacert` всякий раз, когда вы запускаете утилиту `curl`:

```
root@curl:/# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/
          ➥ serviceaccount/ca.crt
```

Сейчас вы можете попасть на сервер API без использования параметра `--cacert`:

```
root@curl:/# curl https://kubernetes
Unauthorized
```

Теперь намного приятнее. Ваш клиент (`curl`) уже доверяет серверу API, но сам сервер API говорит, что вы не авторизованы для доступа к нему, потому что он не знает, кто вы такой.

Аутентификация на сервере API

Для того чтобы он стал читать и даже обновлять и/или удалять объекты API, развернутые в кластере, вам нужно пройти аутентификацию на сервере. Чтобы аутентифицироваться, вам нужен токен аутентификации. К счастью, токен предоставляется с помощью упомянутого ранее секрета default-token и хранится в файле `token` в томе `secret`. Как следует из названия секрета, в этом заключается основная цель секрета.

Вы собираетесь использовать этот токен для доступа к серверу API. Сначала загрузите токен в переменную среды:

```
root@curl:/# TOKEN=$(cat /var/run/secrets/kubernetes.io/
          ➥ serviceaccount/token)
```

Теперь токен хранится в переменной среды `TOKEN`. Его можно использовать при отправке запросов на сервер API. Это показано в следующем ниже листинге.

Листинг 8.13. Получение правильного отклика от сервера API

```
root@curl:/# curl -H "Authorization: Bearer $TOKEN" https://kubernetes
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/apps/v1beta1",
    "/apis/authorization.k8s.io",
    ...
    "/ui/",
    "/version"
  ]
}
```

Деактивация управления ролевым доступом (RBAC)

Если вы используете кластер Kubernetes с активированным RBAC (role-based access control), то учетная запись службы может быть не авторизована для доступа к серверу API. Вы узнаете об учетных записях служб и RBAC в главе 12. На данный момент самый простой способ разрешить запрос к серверу API – это обойти RBAC, выполнив следующую ниже команду:

```
$ kubectl create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \
  --group=system:serviceaccounts
```

Это дает всем учетным записям служб (можно также сказать, всех модулей) привилегии администратора кластера, позволяя им делать все, что они хотят. Очевидно, что делать это опасно и никогда не должно делаться в рабочем окружении. Для целей тестирования это нормально.

Как вы можете видеть, вы передали токен внутри HTTP-заголовка `Authorization` в запросе. Сервер API распознал токен как подлинный и вернул правильный отклик. Теперь можно исследовать все ресурсы в кластере, как это было несколько разделов назад.

Например, можно вывести список всех модулей в одном пространстве имен. Но сначала вам нужно знать, в каком пространстве имен работает модуль `curl`.

Получение пространства имен, в котором выполняется модуль

В первой части этой главы вы видели, как передавать пространство имен модуля через Downward API. Но если вы были внимательны, то, вероятно, заметили, что ваш том `secret` тоже содержит файл с именем `namespace`. Он содержит пространство имен, в котором работает модуль, поэтому, вместо того чтобы явно передавать пространство имен в модуль через переменную среды, вы можете прочитать файл. Загрузите содержимое файла в переменную среды `NS`, а затем выведите список всех модулей, как показано в следующем ниже листинге.

Листинг 8.14. Вывод списка модулей в собственном пространстве имен модуля

```
root@curl:/# NS=$(cat /var/run/secrets/kubernetes.io/
           ➥ serviceaccount/namespace)
root@curl:/# curl -H "Authorization: Bearer $TOKEN"
           ➥ https://kubernetes/api/v1/namespaces/$NS/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
```

И вот результат. Используя три файла в каталоге смонтированного тома `secret`, вы вывели список всех модулей, работающих в том же пространстве имен, что и модуль. Таким же образом можно получить другие объекты API и даже их обновить, отправив вместо простых запросов `GET` запрос `PUT` или `PATCH`.

Краткий обзор того, как модули обмениваются с Kubernetes

Давайте подытожим, каким образом приложение, запущенное в модуле, может получать доступ к API Kubernetes должным образом:

- приложение должно проверить, подписан ли сертификат сервера API центром сертификации, сертификат которого находится в файле ca.crt;
- приложение должно пройти аутентификацию, отправив заголовок `Authorization` с токеном носителя из файла `token`;
- файл `namespace` должен использоваться для передачи пространства имен на сервер API при выполнении операций CRUD над объектами API в пространстве имен модуля.

> **ОПРЕДЕЛЕНИЕ.** CRUD означает создание (Create), чтение (Read), обновление (Update) и удаление (Delete). Соответствующие методы HTTP – это соответственно POST, GET, PATCH/PUT и DELETE.

Все три аспекта модуля относительно обмена с сервером API отображены на рис. 8.5.

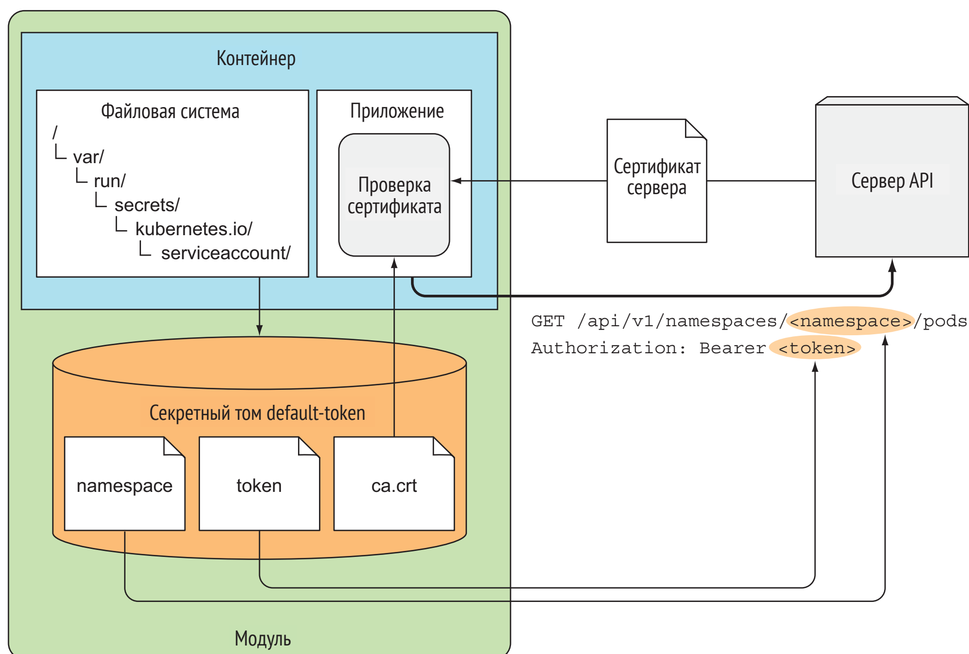


Рис. 8.5. Использование файлов из секрета default-token для обмена с сервером API

8.2.3 Упрощение взаимодействия сервера API с контейнерами-посредниками

Работа с HTTPS, сертификатами и токенами аутентификации иногда кажется разработчикам слишком сложной. Я видел, как разработчики слишком много раз отключали проверку сертификатов сервера (и я признаю, что иногда делаю это сам). К счастью, вы можете сделать взаимодействие намного проще, сохраняя его в безопасности.

Помните команду `kubectl proxy`, которой мы коснулись в разделе 8.2.1? Вы выполнили эту команду на локальном компьютере, чтобы упростить доступ к серверу API. Вместо того чтобы отправлять запросы непосредственно на сервер API, вы отправляете их на прокси-сервер и позволяете ему заботиться об аутентификации, шифровании и серверной верификации. Такой же метод можно использовать внутри ваших модулей тоже.

Знакомство с шаблоном контейнера-посредника

Представьте себе приложение, которое (среди прочего) должно запрашивать сервер API. Вместо того чтобы напрямую обмениваться с сервером API, как вы это делали в предыдущем разделе, вы можете выполнить команду `kubectl proxy` в контейнере-посреднике рядом с главным контейнером и общаться с сервером API через него.

Вместо того чтобы обмениваться с сервером API напрямую, приложение в главном контейнере может подключиться к послу через HTTP (вместо HTTPS) и позволять прокси посредника обрабатывать HTTPS-подключение с сервером API, прозрачно заботясь о безопасности (см. рис. 8.6). Для этого используются файлы из системного токена тома `secret`.

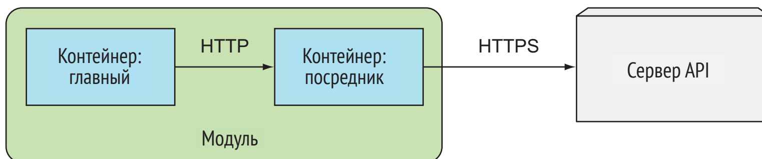


Рис. 8.6. Использование посла для подключения к серверу API

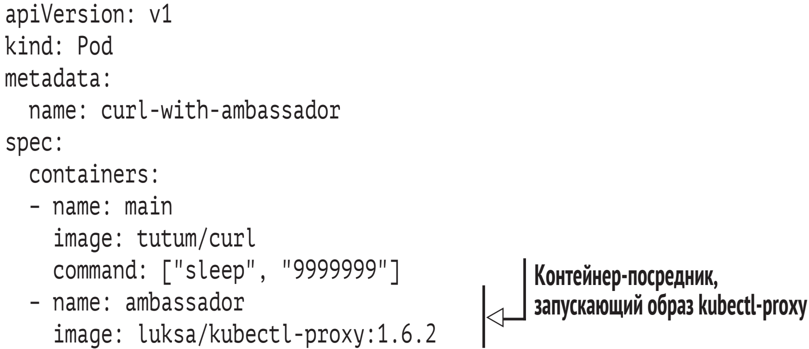
Поскольку все контейнеры делят между собой одинаковый loopback-сетевой интерфейс, ваше приложение может получить доступ к прокси через порт на localhost.

Запуск модуля curl с дополнительным контейнером-послом

Чтобы увидеть шаблон контейнера-посредника в действии, вы создадите новый модуль, подобный ранее созданному модулю `curl`, но на этот раз вместо запуска одного контейнера в модуле вы запустите дополнительный контейнер-посредник, основанный на образе универсального контейнера `kubectl-proxy`, который я создал и отправил в Docker Hub. Если вы хотите создать его самостоятельно, то вы найдете файл Dockerfile для этого образа в архиве кода (в /Chapter 08/kubectl-proxy/).

Манифест модуля показан в следующем ниже листинге.

Листинг 8.15. Модуль с контейнером-послом: curl-with-ambassador.yaml




```
apiVersion: v1
kind: Pod
metadata:
  name: curl-with-ambassador
spec:
  containers:
  - name: main
    image: tutum/curl
    command: ["sleep", "9999999"]
  - name: ambassador
    image: luksa/kubectl-proxy:1.6.2
```

Секция spec модуля почти такая же, как и раньше, но с другим именем модуля и дополнительным контейнером. Запустите модуль и войдите в главный контейнер с помощью команды:

```
$ kubectl exec -it curl-with-ambassador -c main bash
root@curl-with-ambassador:/#
```

Ваш модуль теперь имеет два контейнера, и вам нужно запустить оболочку bash в главном контейнере, следовательно, используется параметр `-c main`. Если вы хотите выполнить эту команду в первом контейнере модуля, то не нужно указывать контейнер явным образом. Но если вы хотите запустить команду в любом другом контейнере, то вам нужно указать имя контейнера с помощью параметра `-c`.

Обмен с сервером API через посредника

Далее вы попробуете подключиться к серверу API через контейнер-посредник. По умолчанию прокси-сервер `kubectl` привязывается к порту 8001, и поскольку оба контейнера в модуле используют одни и те же сетевые интерфейсы, включая loopback, утилиту `curl` можно указать на `localhost:8001`, как показано в следующем ниже листинге.

Листинг 8.16. Доступ к серверу API через контейнер-посол

```
root@curl-with-ambassador:/# curl localhost:8001
{
  "paths": [
    "/api",
    ...
    ]
}
```

Отлично! Результат, напечатанный утилитой `curl`, является тем же откликом, который вы встречали ранее, но на этот раз вам не нужно было иметь дело с токенами аутентификации и сертификатами сервера.

Чтобы получить четкое представление о том, что именно произошло, обратитесь к рис. 8.7. Утилита `curl` отправила простой HTTP-запрос (без каких-либо заголовков аутентификации) на прокси-сервер, работающий внутри контейнера-посла, а затем прокси отправил HTTPS-запрос на сервер API, обработав аутентификацию клиента, отправив токен и проверив удостоверение сервера путем проверки его сертификата.

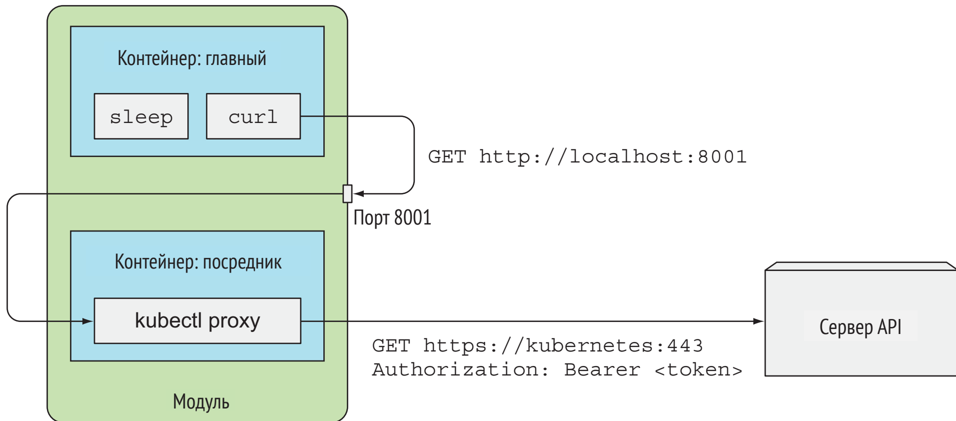


Рис. 8.7. Передача работы по шифрованию, аутентификации и проверке сервера прокси-серверу kubectl proxy в контейнере-посреднике

Это отличный пример того, как контейнер-посредник может использоваться для того, чтобы скрыть сложности подключения к внешней службе и упростить работу приложения в главном контейнере. Контейнер-посредник можно использовать в различных приложениях, независимо от того, на каком языке написано главное приложение. Его недостатком является то, что выполняется дополнительный процесс, который потребляет дополнительные ресурсы.

8.2.4 Использование клиентских библиотек для обмена с сервером API

Если приложению требуется выполнить лишь несколько простых операций на сервере API, часто можно обойтись обычной клиентской библиотекой HTTP и выполнять простые HTTP-запросы, в особенности если использовать контейнер-посредник `kubectl-proxy` так, как это было продемонстрировано в предыдущем примере. Но если вы планируете делать больше, чем простые запросы API, то лучше использовать одну из существующих клиентских библиотек API Kubernetes.

Использование существующих клиентских библиотек

В настоящее время существуют две клиентские библиотеки API Kubernetes, поддерживаемые специальной группой по направлениям API Machinery (SIG):

- клиент Golang – https://github.com/kubernetes/client-go;
- Python – https://github.com/kubernetes/client-go.

ПРИМЕЧАНИЕ. Сообщество Kubernetes имеет ряд специальных групп по направлениям (special interest group, SIG) и рабочих групп, которые сосредоточены на конкретных частях экосистемы Kubernetes. Вы найдете их список по адресу https://github.com/kubernetes/community/blob/master/sig-list.md.

В дополнение к двум официально поддерживаемым библиотекам ниже приведен список пользовательских клиентских библиотек для многих других языков:

- клиент Java от Fabric8 – https://github.com/fabric8io/kubernetes-client;
- клиент Java от Amdatu – https://bitbucket.org/amdatulabs/amdatu-kubernetes;
- клиент Node.js от tenxcloud – https://github.com/tenxcloud/node-kubernetes-client;
- клиент Node.js от GoDaddy – https://github.com/godaddy/kubernetes-client;
- клиент PHP – https://github.com/devstub/kubernetes-api-php-client;
- еще один клиент PHP – https://github.com/maclof/kubernetes-client;
- Ruby – https://github.com/Ch00k/kubr;
- еще один клиент Ruby – https://github.com/abonas/kubeclient;
- Clojure – https://github.com/yanatan16/clj-kubernetes-api;
- Scala – https://github.com/doriordan/skuber;
- Perl – https://metacpan.org/pod/Net::Kubernetes.

Эти библиотеки обычно поддерживают HTTPS и берут на себя аутентификацию, поэтому вам не придется использовать контейнер-посредник.

Пример взаимодействия с Kubernetes с помощью клиента Java Fabric8

Чтобы получить представление о том, как клиентские библиотеки позволяют взаимодействовать с сервером API, в следующем ниже листинге показан пример вывода списка служб в приложении Java с помощью клиента Fabric8 Kubernetes.

Листинг 8.17. Вывод списка, создание, обновление и удаление модулей с помощью Java-клиента Fabric8

```
import java.util.Arrays;
import io.fabric8.kubernetes.api.model.Pod;
import io.fabric8.kubernetes.api.model.PodList;
```

```
import io.fabric8.kubernetes.client.DefaultKubernetesClient;
import io.fabric8.kubernetes.client.KubernetesClient;
public class Test {
  public static void main(String[] args) throws Exception {
    KubernetesClient client = new DefaultKubernetesClient();

    // list pods in the default namespace
    PodList pods = client.pods().inNamespace("default").list();
    pods.getItems().stream()
      .forEach(s -> System.out.println("Found pod: " +
               s.getMetadata().getName()));

    // create a pod
    System.out.println("Creating a pod");
    Pod pod = client.pods().inNamespace("default")
      .createNew()
      .withNewMetadata()
        .withName("programmatically-created-pod")
      .endMetadata()
      .withNewSpec()
        .addNewContainer()
          .withName("main")
          .withImage("busybox")
          .withCommand(Arrays.asList("sleep", "99999"))
        .endContainer()
      .endSpec()
      .done();
    System.out.println("Created pod: " + pod);

    // edit the pod (add a label to it)
    client.pods().inNamespace("default")
      .withName("programmatically-created-pod")
      .edit()
      .editMetadata()
        .addToLabels("foo", "bar")
      .endMetadata()
      .done();
    System.out.println("Added label foo=bar to pod");

    System.out.println("Waiting 1 minute before deleting pod...");
    Thread.sleep(60000);

    // delete the pod
    client.pods().inNamespace("default")
      .withName("programmatically-created-pod")
      .delete();
```

```
    System.out.println("Deleted the pod");
  }
}
```

Приведенный программный код говорит сам за себя, в особенности потому, что клиент Fabric8 предоставляет хороший и гибкий предметно-ориентированный язык (DSL) API, который легко читать и понимать.

Создание собственной библиотеки с помощью Swagger и OpenAPI

Если клиент на выбранном вами языке программирования отсутствует, то вы можете использовать платформу API Swagger для создания клиентской библиотеки и документации. Сервер API Kубернetes предоставляет определения API Swagger в /swaggerapi и спецификацию OpenAPI в /swagger.json.

Для того чтобы узнать о платформе Swagger больше, посетите веб-сайт по адресу `http://swagger.io`.

Исследование API с помощью Swagger UI

Ранее в главе я упомянул, что покажу вам более оптимальный способ исследования API REST, вместо того чтобы попадать на конечные точки REST с помощью утилиты `curl`. Платформа Swagger, о которой я упоминал в предыдущем разделе, – это не только инструмент для описания API, но и веб-интерфейс для исследования интерфейсов API REST, если они предоставляют доступ к определениям API Swagger. Более оптимальный способ исследования API REST подразумевает использование этого пользовательского интерфейса.

Kubernetes не только предоставляет доступ к API Swagger, но также имеет интерфейс Swagger, интегрированный в сервер API, хотя он не активирован по умолчанию. Его можно активировать, запустив сервер API с помощью параметра `--enable-swagger-ui=true`.

> **СОВЕТ.** Если вы используете Minikube, то можете активировать интерфейс Swagger при запуске кластера: `minikube start --extra-config=apiserver.Features.EnableSwaggerUI=true`.

После активирования пользовательского интерфейса его можно открыть в браузере, направив его на:

```
http(s)://<сервер api>:<порт>/swagger-ui
```

Я призываю вас попробовать Swagger UI. Он не только позволяет просматривать API Kubernetes, но и взаимодействовать с ним (например, можно отправлять манифесты ресурсов в формате JSON, исправлять или удалять ресурсы).

8.3 Резюме

После прочтения этой главы вы теперь знаете, как ваше приложение, запущенное в модуле, может получать данные о себе, о других модулях и о других компонентах, развернутых в кластере. Вы узнали:

- как имя модуля, пространство имен и другие метаданные могут быть доступны процессу через переменные среды или файлы в томе `downwardAPI`;
- как запросы и лимиты на ЦП и память передаются в ваше приложение в любой единице измерения, которую требует ваше приложение;
- как модуль может использовать тома `downwardAPI` для получения актуальных метаданных, которые могут изменяться в течение срока службы модуля (например, метки и аннотации);
- каким образом можно просматривать API REST Kubernetes с помощью команды `kubectl proxy`;
- как модули могут найти местоположение сервера API с помощью переменных среды или DNS, подобно любой другой службе, определенной в Kubernetes;
- как работающее в модуле приложение может проверить, что оно взаимодействует с сервером API, и как оно может аутентифицироваться;
- как использование контейнера-посла может намного упростить обмен с сервером API изнутри приложения;
- как клиентские библиотеки способны помочь вам взаимодействовать с Kubernetes за считанные минуты.

В этой главе вы познакомились с тем, как обмениваться с сервером API, поэтому следующий шаг – узнать больше о том, как он работает. Вы сделаете это в главе 11, но прежде чем мы углубимся в такие детали, вам все равно нужно узнать о двух других ресурсах Kubernetes – Deployment и StatefulSet. Они объясняются в следующих двух главах.

Глава 9

Развертывания: декларативное обновление приложений

Эта глава посвящена:

- замене модулей на более новые версии;
- обновлению управляемых модулей;
- декларативному обновлению модулей с помощью ресурсов развертывания;
- выполнению плавных обновлений;
- автоматическому блокированию развертки плохих версий;
- управлению скоростью развертывания;
- возврату модулей к предыдущей версии.

Теперь вы знаете, как упаковывать компоненты приложения в контейнеры, группировать их в модули, предоставить им временное или постоянное хранилище, передавать им как секретные, так и несекретные конфигурационные данные и разрешать модулям находить друг друга и взаимодействовать. Вы знаете, как запускать полноценную систему, состоящую из самостоятельно работающих мелких компонентов – микросервисов, если захотите. Есть что-то еще?

В перспективе вы захотите обновить свое приложение. В этой главе описывается, как обновлять приложения, запущенные в кластере Kubernetes, и как Kubernetes помогает переходить к процессу обновления с нулевым временем простоя. Хотя этого можно достичь, используя только контроллеры репликации ReplicationController или наборы реплик ReplicaSet, Kubernetes также предоставляет ресурс развертывания Deployment, который опирается на ресурсы ReplicaSet и позволяет выполнять декларативные обновления приложений. Если вы не совсем уверены, что это значит, продолжайте читать – это не так сложно, как кажется.

9.1 Обновление приложений, работающих в модулях

Начнем с простого примера. Представьте себе набор экземпляров модуля, предоставляющих службу другим модулям и/или внешним клиентам. Прочитав данную книгу до этого момента, вы, вероятно, поймете, что эти модули созданы с помощью ReplicationController или ReplicaSet. Существует также служба, через которую клиенты (приложения, работающие в других модулях или внешних клиентах) получают доступ к модулям. Вот как выглядит базовое приложение в Kubernetes (см. рис. 9.1).

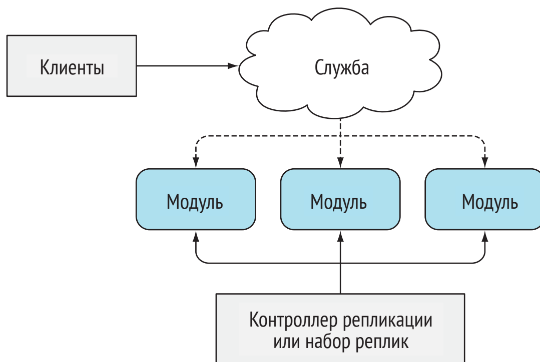


Рис. 9.1. Схема базового приложения, работающего в Kubernetes

Первоначально модули выполняют первую версию вашего приложения – предположим, что его образ помечен как v1. Затем вы разрабатываете новую версию приложения и отправляете ее в репозиторий образов в качестве нового образа, помеченного как v2. Вы хотели бы заменить все модули этой новой версией. Поскольку вы не можете изменить образ существующего модуля после его создания, вам нужно удалить старые модули и заменить их новыми, запустив новый образ.

У вас есть два способа обновить все эти модули. Можно выполнить одно из следующих действий:

- сначала удалить все существующие модули, а затем запустить новые;
- начать новые и, как только они запустятся, удалить старые. Вы можете сделать это либо за один раз путем добавления всех новых модулей, а затем удаления всех старых, либо последовательно путем постепенного добавления новых модулей и удаления старых.

Обе эти стратегии имеют свои преимущества и недостатки. Первый вариант приведет к короткому периоду времени, когда приложение будет недоступно. Второй вариант требует, чтобы приложение могло работать в ситуации, когда две версии приложения запущены одновременно. Если приложение хранит данные в хранилище данных, новая версия не должна изменять схему данных или сами данные таким образом, чтобы нарушать предыдущую версию.

Как выполнить эти два метода обновления в Kubernetes? Прежде всего давайте посмотрим, как это сделать вручную; затем, как только вы узнаете, что участвует в этом процессе, вы познакомитесь с тем, как заставить Kubernetes выполнять обновление автоматически.

9.1.1 Удаление старых модулей и замена их новыми

Вы уже знаете, как заставить контроллер репликации заменить все экземпляры его модуля на модули, выполняющие новую версию. Вероятно, вы помните, что шаблон модуля контроллера репликации можно обновлять в любое время. Когда контроллер репликации создает новые экземпляры, для их создания он использует обновленный шаблон модуля.

Если у вас есть контроллер репликации, управляющий набором модулей версии v1, вы можете легко заменить их, изменив шаблон модуля, чтобы он ссылался на версию v2 образа, а затем удалить старые экземпляры модуля. Контроллер репликации заметит, что никакие модули не совпадают с его селектором меток, и это приведет к развертыванию новых экземпляров. Весь процесс показан на рис. 9.2.

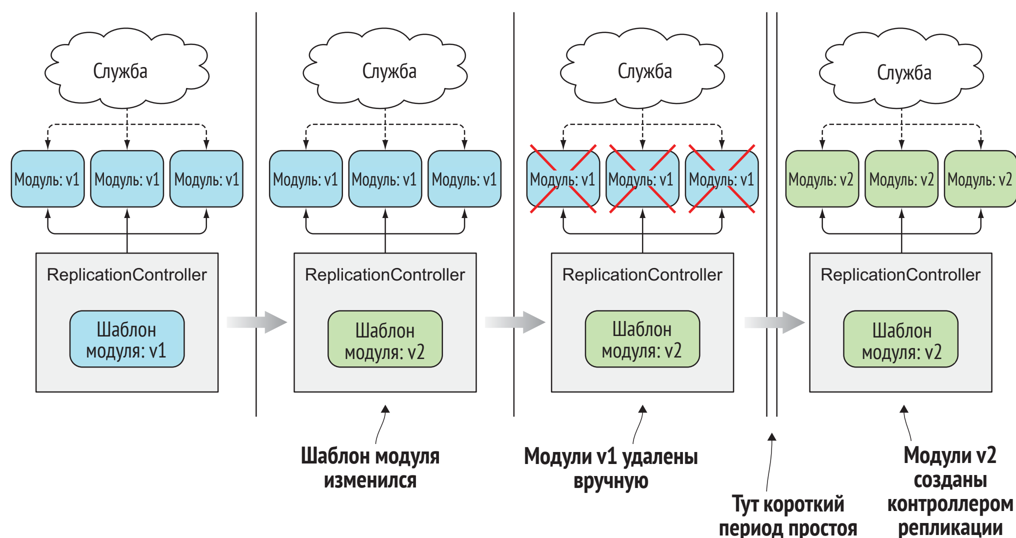


Рис. 9.2. Обновление модулей путем изменения шаблона модуля контроллера репликации и удаления старых модулей

Это самый простой способ обновить набор модулей, в случае если вы можете позволить короткое время простоя между временем удаления старых модулей и запуском новых.

9.1.2 Запуск новых модулей, а затем удаление старых

Если вы не хотите допускать время простоя и ваше приложение поддерживает одновременный запуск нескольких версий, то вы можете сделать наоборот и сначала запустить все новые модули и только потом удалить старые. Это

потребует больше аппаратных ресурсов, потому что вы будете иметь двойное количество модулей, работающих в одно и то же время в течение короткого времени.

Это немного более сложный метод, по сравнению с предыдущим, но вы сможете это сделать, если объедините то, что вы до этого узнали о контроллерах репликации и службах.

Переход сразу со старой версии на новую

Модули обычно предваряются службой. Можно сделать так, чтобы служба предваряла только первоначальную версию модулей до тех пор, пока вы поднимаете модули, реализующие новую версию. Затем, как только все новые модули подняты, вы можете изменить селектор меток службы и переключить службу на новые модули, как показано на рис. 9.3. Это называется *сине-зеленым развертыванием*. После переключения, и как только вы будете уверены, что новая версия функционирует правильно, вы можете удалить старые модули, удалив старый контроллер репликации.

> **ПРИМЕЧАНИЕ.** Селектор модулей службы можно изменить с помощью команды `kubectl set selector`.

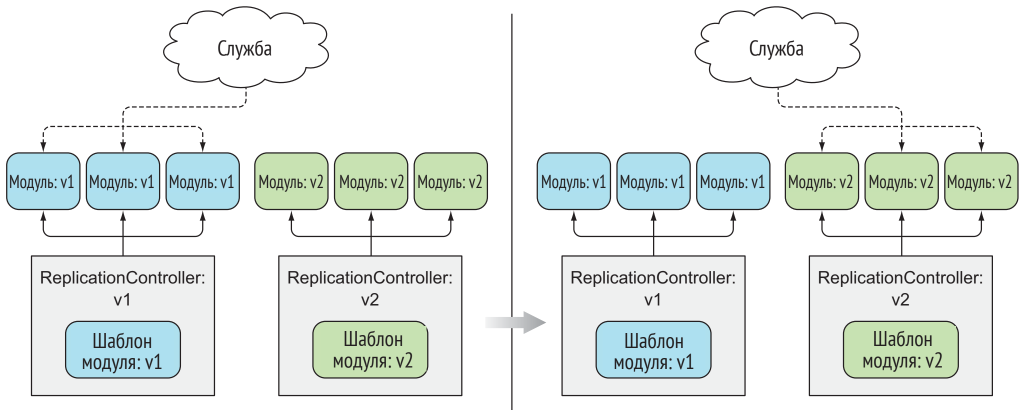


Рис. 9.3. Переключение службы со старых модулей на новые

Выполнение скользящего обновления

Вместо того чтобы сразу поднимать все новые модули и удалять старые, вы также можете выполнить плавное обновление, которое заменяет модули шаг за шагом. Это можно сделать путем медленного уменьшения масштаба предыдущего контроллера репликации и увеличения масштаба нового. В этом случае вам нужно, чтобы селектор модулей службы включал как старые, так и новые модули, поэтому он направляет запросы к обоим наборам модулей (рис. 9.4).

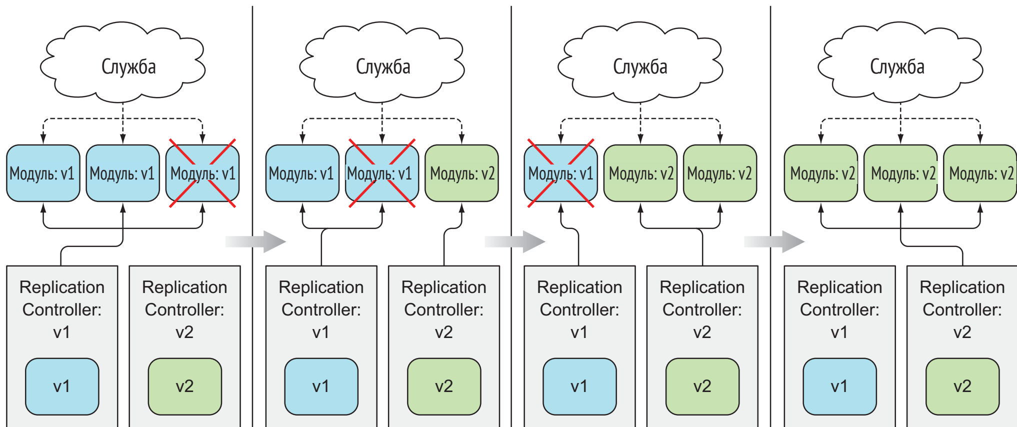


Рис. 9.4. Обновление модулей с помощью двух контроллеров репликации

Выполнение плавного обновления вручную трудоемко и подвержено ошибкам. В зависимости от количества реплик для выполнения процесса обновления необходимо выполнить несколько команд в правильном порядке. К счастью, Kubernetes позволяет выполнять обновление с помощью одной команды. Вы узнаете, как это сделать, в следующем разделе.

9.2 Выполнение автоматического плавного обновления с помощью контроллера репликации

Вместо ручного выполнения скользящего обновления, используя контроллеры репликации, вы можете дать это сделать инструменту `kubectl`. Использование `kubectl` для выполнения обновления значительно упрощает процесс, но, как вы увидите позже, теперь это устаревший способ обновления приложений. Тем не менее сначала мы рассмотрим именно данный вариант, потому что этот способ был исторически самым первым способом выполнения автоматического обновления, и к тому же он позволяет нам обсудить этот процесс, не вводя слишком много дополнительных понятий.

9.2.1 Запуск первоначальной версии приложения

Очевидно, что перед обновлением приложения необходимо развернуть приложение. В качестве начальной версии вы будете использовать слегка измененную версию приложения kubia NodeJS, созданную в главе 2. В случае если вы не помните, что оно делает, то это простое веб-приложение, которое в ответе HTTP возвращает хостнейм модуля.

Создание приложения v1

Вы измените приложение так, чтобы оно в ответе также возвращало номер версии, что позволит вам различать разные версии, которые вы собираетесь

создавать. Я уже собрал и отправил образ приложения в реестр Docker Hub под `luksa/kubia:v1`. Ниже приведен код приложения.

Листинг 9.1. Версия v1 нашего приложения: v1/app.js

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");
var handler = function(request, response) {
  console.log("Received request from " + request.connection.remoteAddress);
  response.writeHead(200);
  response.end("This is v1 running in pod " + os.hostname() + "\n");
};
var www = http.createServer(handler);
www.listen(8080);
```

Запуск приложения и предоставление к нему доступа через службу с помощью одного файла YAML

Для того чтобы запустить приложение, необходимо создать контроллер репликации и службу балансировки нагрузки. Это позволяет получать доступ к приложению извне. На этот раз, вместо того чтобы создавать эти два ресурса по отдельности, вы создадите один файл YAML для них обоих и опубликуете его в API Kubernetes с помощью одной команды `kubectl create`. Манифест YAML может содержать несколько объектов, разделенных строкой, содержащей три дефиса, как показано в следующем ниже листинге.

Листинг 9.2. YAML, содержащий контроллер репликации и службу: kubia-rc-and-service-v1.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia-v1
spec:
  replicas: 3
  template:
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
      - image: luksa/kubia:v1
        name: nodejs
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  type: LoadBalancer
  selector:
    app: kubia
  ports:
  - port: 80
    targetPort: 8080
```

Файлы YAML могут содержать несколько определений ресурсов, разделенных строкой с тремя дефисами

Служба поддерживает все модули, созданные контроллером репликации

YAML задает контроллер репликации под названием `kubia-v1` и службу под названием `kubia`. Продолжайте и отправьте YAML в Kubernetes. Через некоторое время все три модуля `v1` и подсистема балансировки нагрузки должны быть запущены, и вы сможете найти внешний IP-адрес службы и начать попадать в службу с помощью утилиты `curl`, как показано в следующем ниже листинге.

Листинг 9.3. Получение внешнего IP-адреса службы и попадание в службу в цикле с помощью утилиты curl

```
$ kubectl get svc kubia
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubia 10.3.246.195 130.211.109.222 80:32143/TCP 5m
$ while true; do curl http://130.211.109.222; done
This is v1 running in pod kubia-v1-qr192
This is v1 running in pod kubia-v1-kbtsk
This is v1 running in pod kubia-v1-qr192
This is v1 running in pod kubia-v1-2321o
...
```

> **ПРИМЕЧАНИЕ.** Всли вы используете Minikube или любой другой кластер Kubernetes, где службы балансировки нагрузки не поддерживаются, то для доступа к приложению вы можете применить порт узла службы. Это разъясняется в главе 5.

9.2.2 Выполнение плавного обновления с помощью kubectl

Далее вы создадите версию 2 приложения. Чтобы все оставалось простым, вам нужно лишь изменить отклик, который будет сообщать: «This is v2»:

```
response.end("This is v2 running in pod " + os.hostname() + "\n");
```

Эта новая версия доступна в образе `luksa/kubia: v2` в Docker Hub, поэтому вам не нужно создавать ее самостоятельно.

Внесение изменений в тег того же самого образа

Изменение приложения и внесение изменений в тег одного и того же образа не является хорошей идеей, но мы все, как правило, это делаем во время разработки. Если вы модифицируете тег `latest`, это не проблема, но когда вы тегируете образ другим тегом (например, тегом `v1` вместо `latest`), после того как образ скачан узлом, образ будет храниться на узле и больше не будет скачиваться, когда запускается новый модуль, использующий тот же самый образ (по крайней мере, эта политика принята по умолчанию для выгрузки образов из хранилищ).

Это означает, что любые изменения, внесенные в образ, не будут подхвачены, если вы отправите их на тот же тег. Если новый модуль назначен на тот же самый узел, Kubelet запустит старую версию образа. С другой стороны, узлы, которые не выполняли старую версию, скачают и выполнят новый образ, поэтому в итоге вы можете оказаться с двумя разными версиями работающего модуля. Чтобы этого не происходило, необходимо для свойства контейнера `imagePullPolicy` задать значение `Always`.

Вам следует осознавать, что принятая по умолчанию политика выгрузки образа `imagePullPolicy` зависит от тега образа. Если контейнер ссылается на тег `latest` (явно или не указывая тег вообще), по умолчанию `imagePullPolicy` равняется `Always`, но если же контейнер ссылается на любой другой тег, то по умолчанию политика равняется `IfNotPresent`.

Если при использовании тега, отличного от `latest`, вы отправляете изменения в образ без изменения тега, то вам нужно должным образом установить политику `imagePullPolicy`. Или еще лучше, убедитесь, что вы всегда отправляете изменения в образ под новым тегом.

Оставьте цикл `curl` работающим и откройте еще один терминал, где вы запустите плавное обновление. Для выполнения обновления выполните команду `kubectl rolling-update`. Вам нужно лишь сообщить, какой контроллер репликации вы заменяете, дать имя новому контроллеру репликации и указать новый образ, которым вы хотели бы заменить исходный. В следующем ниже листинге показана полная команда для выполнения скользящего обновления.

Листинг 9.4. Инициализация скользящего обновления rolling-update контроллера репликации с использованием команды kubectl

```
$ kubectl rolling-update kubia-v1 kubia-v2 --image=luksa/kubia:v2
Created kubia-v2
Scaling up kubia-v2 from 0 to 3, scaling down kubia-v1 from 3 to 0 (keep 3
    pods available, don't exceed 4 pods)
...
```

Поскольку вы меняете контроллер репликации kubia-v1 на тот, который работает с версией 2 приложения kubia, вы хотите назвать новый контроллер репликации kubia-v2 и использовать образ контейнера luksa/kubia:v2.

При выполнении команды немедленно создается новый контроллер репликации kubia-v2. Состояние системы на данный момент показано на рис. 9.5.

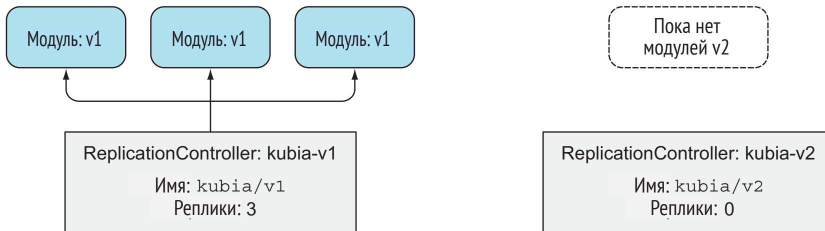


Рис. 9.5. Состояние системы сразу после запуска обновления

Как показано в следующем ниже листинге, шаблон модуля нового контроллера репликации ссылается на образ lukas/kuba:v2, а его начальное число требуемых реплик равняется 0.

Листинг 9.5. Описание нового контроллера репликации, созданного с помощью скользящего обновления

```
$ kubectl describe rc kubia-v2
Name:         kubia-v2
Namespace:    default
Image(s):     luksa/kubia:v2       <- Новый контроллер репликации ссылается на образ версии 2
Selector:     app=kubia,deployment=757d16a0f02f6a5c387f2b5edb62b155
Labels:       app=kubia
Replicas:     0 current / 0 desired  <- Изначально требуемое количество реплик равно нулю
...
```

Действия, выполняемые kubectl до начала плавного обновления

Инструмент kubectl создал этот контроллер репликации путем копирования контроллера kubia-v1 и изменения образа в его шаблоне модуля. Если вы внимательно посмотрите на селектор меток контроллера, то вы заметите, что он тоже был изменен. Он включает не только простую метку app=kubia, но и дополнительную метку развертывания deployment, которую модули должны иметь, для того чтобы управляться этим контроллером репликации.

Вы, вероятно, уже знаете об этом, но это необходимо для того, чтобы новые и старые контроллеры репликации не работали на одном наборе модулей. Однако даже если модули, созданные новым контроллером, имеют еще одну метку deployment в дополнение к метке app=kubia, значит ли это, что они будут выбраны селектором первого контроллера репликации, так как у него задано значение app=kubia?

Да, это именно то, что произойдет, но есть один подвох. Процесс плавного обновления rolling-update также изменил селектор первого контроллера репликации:

```
$ kubectl describe rc kubia-v1
Name:        kubia-v1
Namespace:   default
Image(s):    luksa/kubia:v1
Selector:    app=kubia,deployment=3ddd307978b502a5b975ed4045ae4964-orig
```

Хорошо, но не означает ли это, что первый контроллер теперь видит ноль модулей, совпадающих с его селектором, потому что ранее созданные им три модуля содержат метку `app=kubia`? Нет, потому что непосредственно перед изменением селектора контроллера репликации `kubectl` также изменил метки действующих модулей:

```
$ kubectl get po --show-labels
NAME            READY  STATUS   RESTARTS  AGE  LABELS
kubia-v1-m33mv  1/1    Running  0         2m   app=kubia,deployment=3ddd...
kubia-v1-nmzw9  1/1    Running  0         2m   app=kubia,deployment=3ddd...
kubia-v1-cdtey  1/1    Running  0         2m   app=kubia,deployment=3ddd...
```

Если это становится слишком сложным, изучите рис. 9.6, который показывает модули, их метки и два контроллера репликации наряду с их селекторами модулей.

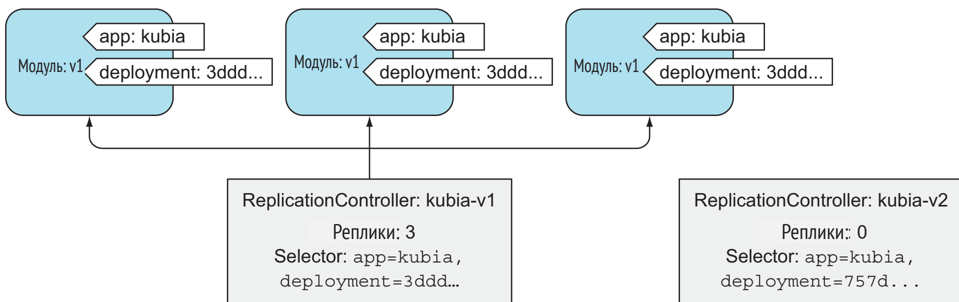


Рис. 9.6. Подробное состояние старого и нового контроллеров репликации и модулей в начале скользящего обновления

Инструменту `kubectl` пришлось все это сделать, прежде чем даже начать увеличивать или уменьшать масштаб чего-либо. Теперь представьте, что вы делаете скользящее обновление вручную. Здесь легко увидеть, что вы совершаете ошибку и, возможно, позволяете контроллеру репликации убить все ваши модули – модули, которые активно обслуживают ваших производственных клиентов!

Замена старых модулей новыми путем масштабирования двух контроллеров репликации

Выполнив всю эту настройку, kubectl начинает замену модулей, сначала увеличивая масштаб нового контроллера до 1. Этот контроллер, следовательно, создает первый модуль версии v2. Затем kubectl уменьшает масштаб старого контроллера репликации на 1. Это показано в следующих двух строках, напечатанных инструментом kubectl:

```
Scaling kubia-v2 up to 1
Scaling kubia-v1 down to 2
```

Поскольку служба нацелена на все модули с меткой app=kubia, вы увидите, что ваши запросы утилиты curl перенаправляются на новый модуль версии v2 каждые несколько итераций цикла:

```
This is v2 running in pod kubia-v2-nmzw9   <-
This is v1 running in pod kubia-v1-kbtsk
This is v1 running in pod kubia-v1-2321o
This is v2 running in pod kubia-v2-nmzw9   <-
...
```

Запросы попадают на модуль, выполняющий новую версию

На рис. 9.7 показано текущее состояние системы.

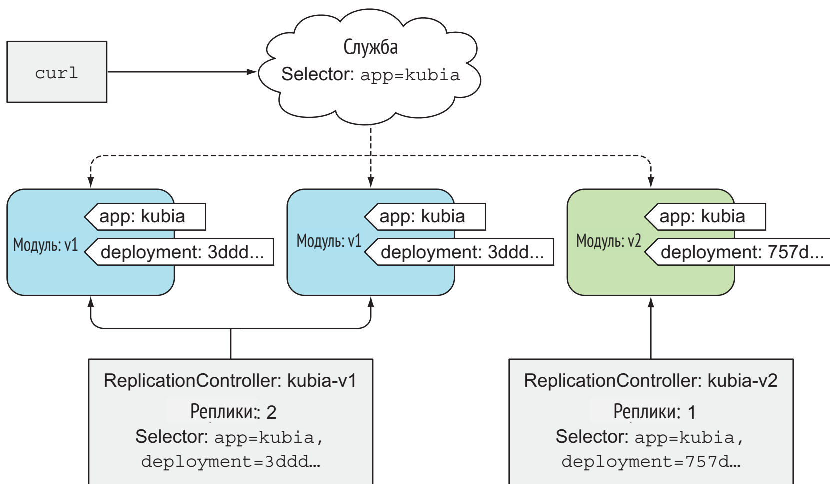


Рис. 9.7. Во время обновления служба перенаправляет запросы и на старые, и на новые модули

По мере того как kubectl продолжает скользящее обновление, вы начинаете наблюдать все больший процент запросов, попадающих в модули v2, так как процесс обновления удаляет больше модулей v1 и заменяет их теми, которые выполняют ваш новый образ. В конечном итоге масштаб исходного контроллера репликации уменьшается до нуля, что приводит к удалению последнего модуля v1, то есть теперь служба будет поддерживаться только модулями v2.

На этом этапе kubectl удалит исходный контроллер репликации, и процесс обновления будет завершен, как показано в следующем ниже листинге.

Листинг 9.6. Заключительные шаги, выполняемые командой kubectl rolling-update

```
...
Scaling kubia-v2 up to 2
Scaling kubia-v1 down to 1
Scaling kubia-v2 up to 3
Scaling kubia-v1 down to 0
Update succeeded. Deleting kubia-v1
replicationcontroller "kubia-v1" rolling updated to "kubia-v2"
```

Теперь у вас остался только контроллер репликации kubia-v2 и три модуля v2. На протяжении всего этого процесса обновления вы попадали на свою службу и всякий раз получали отклик. Фактически вы выполнили плавное обновление с нулевым временем простоя.

9.2.3 Почему плавное обновление kubectl rolling-update устарело

В начале этого раздела я отметил, что имеется еще более оптимальный способ делать обновления, чем посредством kubectl rolling-update. Что такого плохого в этом процессе, что пришлось внедрить более оптимальный?

Дело вот в чем. Для начала я, например, не люблю, когда Kubernetes изменяет объекты, которые я создал. Да, все прекрасно, когда планировщик назначает узел моим модулям, после того как я их создаю, но вот когда Kubernetes изменяет метки моих модулей и селекторы меток моих контроллеров репликации – это то, чего я не ожидаю, и может заставить меня ходить по офису, покрикивая на своих коллег: «Кто возился с моими контроллерами?!?» Но что еще более важно, если вы обратили пристальное внимание на слова, которые я использовал, то вы, вероятно, заметили, что все это время я четко заявлял, что именно клиент kubectl выполнял все эти шаги скользящего обновления.

Это можно увидеть, включив подробное ведение лога с помощью параметра --v во время инициирования плавного обновления:

```
$ kubectl rolling-update kubia-v1 kubia-v2 --image=luksa/kubia:v2 --v 6
```

> **СОВЕТ.** При использовании параметра --v 6 уровень ведения журнала увеличивается настолько, что можно увидеть запросы, отправляемые клиентом kubectl на сервер API.

С помощью этого параметра kubectl будет распечатывать каждый HTTP-запрос, отправляемый на сервер API Kubernetes. Вы увидите запросы PUT на

```
/api/v1/namespaces/default/replicationcontrollers/kubia-v1
```

который является RESTful URL-адресом, представляющим ресурс контроллера репликации `kubia-v1`. Как раз эти запросы и уменьшают масштаб вашего контроллера репликации, а это свидетельствует о том, что именно `kubectl` выполняет масштабирование, вместо того чтобы этим занимался ведущий узел Kubernetes.

> **СОВЕТ.** Воспользуйтесь параметром подробного журналирования при выполнении других команд `kubectl`. Это позволит больше узнать о взаимодействии между `kubectl` и сервером API.

Но что плохого в том, что процесс обновления выполняется на клиенте, а не на сервере? Дело в том, что в вашем случае обновление прошло гладко, но что, если во время выполнения обновления агентом `kubectl` вы потеряли подключение к сети? Процесс обновления будет прерван в середине пути. Модули и контроллеры репликации в итоге окажутся в промежуточном состоянии.

Еще одна причина, почему выполнение такого рода обновления не так хорошо, как оно могло быть, заключается в том, что оно императивное. На протяжении всей этой книги я подчеркивал: Kubernetes связан с тем, что вы сообщаете ему желаемое состояние системы, а Kubernetes достигает этого состояния самостоятельно, выясняя для этого самый лучший способ. Именно так модули разворачиваются, и именно так масштаб модулей увеличивается и уменьшается. Вы никогда не говорите Kubernetes, что нужно добавить дополнительный модуль или удалить лишний – вы изменяете количество требуемых реплик, и все.

Схожим образом вы также захотите изменить требуемый тег образа в определениях модулей, и Kubernetes заменит модули на новые, выполняющие новый образ. Именно это и привело к появлению нового ресурса под названием развертывание Deployment, который теперь является предпочтительным способом развертывания приложений в Kubernetes.

9.3 Использование развертываний для декларативного обновления приложений

Развертывание Deployment – это ресурс более высокого уровня, предназначенный для развертывания приложений и их обновления декларативным образом, вместо того чтобы делать это через контроллер репликации или набор реплик, которые рассматриваются как более низкоуровневые понятия.

Когда вы создаете развертывание, под поверхностью создается ресурс ReplicaSet (а в итоге несколько таких ресурсов). Как вы помните из главы 4, наборы реплик – это новое поколение контроллеров репликации, и следует использовать их. Наборы репликаций точно так же реплицируют и управляют модулями. При использовании развертывания фактические модули создаются и управляются наборами реплик объекта Deployment, а не непосредственно объектом Deployment (связь показана на рис. 9.8).

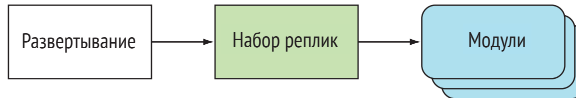


Рис. 9.8. Развертывание поддерживается набором реплик, который контролирует модули развертывания

Вы можете задаться вопросом, зачем усложнять ситуацию, вводя еще один объект поверх контроллера репликации или набора реплик, когда их достаточно, чтобы поддерживать набор экземпляров модуля. Как показывает пример плавного обновления в разделе 9.2, при обновлении приложения необходимо вводить дополнительный контроллер репликации и координировать два контроллера, чтобы они могли танцевать вокруг друг друга, не наступая друг другу на ноги. И поэтому требуется что-то, что координирует этот танец. Ресурс Deployment заботится именно об этом (не сам ресурс развертывания, а процесс контроллера, запущенный в плоскости управления Kubernetes, который это делает; но мы перейдем к этому в главе 11).

Как вы увидите на следующих нескольких страницах, использование развертывания вместо более низкоуровневых конструкций значительно упрощает обновление приложения, поскольку вы определяете требуемое состояние с помощью одного-единственного ресурса развертывания и позволяете Kubernetes позаботиться об остальном.

9.3.1 Создание развертывания

Создание развертывания не отличается от создания контроллера репликации. Развертывание также состоит из селектора меток, требуемого количества реплик и шаблона модуля. Кроме того, оно тоже содержит поле, в котором указывается стратегия развертывания, определяющая способ выполнения обновления при изменении ресурса развертывания.

Создание манифеста развертывания

Давайте посмотрим, как использовать более ранний пример контроллера репликации `kubia-v1` из этой главы и изменить его так, чтобы он описывал развертывание вместо контроллера репликации. Как вы увидите, это требует всего трех тривиальных изменений. Ниже приведен измененный YAML.

Листинг 9.7. Определение развертывания: kubia-deployment-v1.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
```

Развертывание в группе API apps, версии v1beta1

Вы изменили вид с контроллера репликации на развертывание

Нет необходимости включать версию в имя развертывания

```
    name: kubia
    labels:
      app: kubia
  spec:
    containers:
    - image: luksa/kubia:v1
    name: nodejs
```

> **ПРИМЕЧАНИЕ.** Вы найдете более старую версию ресурса развертывания в `extensions/v1beta1` и более новую в `apps/v1beta2` с разными требуемыми полями и разными значениями, выбранными по умолчанию. Имейте в виду, что `kubectl explain` показывает более старую версию.

Поскольку контроллер репликации до этого управлял определенной версией модулей, вы назвали его `kubia-v1`. С другой стороны, развертывание находится выше этой версии. В определенный момент времени развертывание может иметь несколько версий модулей, работающих под его крылом, поэтому его имя не должно ссылаться на версию приложения.

Создание ресурса развертывания

Перед созданием этого развертывания убедитесь, что вы удалили все запущенные контроллеры репликации и модули, но сохранили службу `kubia`. Для того чтобы удалить все эти контроллеры репликации, вы можете использовать переключатель `--all`:

```
$ kubectl delete rc --all
```

Теперь вы готовы к созданию развертывания:

```
$ kubectl create -f kubia-deployment-v1.yaml --record
deployment "kubia" created
```

> **СОВЕТ.** Не забудьте при его создании включить параметр командной строки `--record`. Это позволит записать данную команду в истории ревизий, которая будет полезна позже.

Вывод статуса выкладки развертывания

Для просмотра подробных сведений о развертывании вы можете применить обычные команды `kubectl get deployment` и `kubectl describe deployment`, но здесь я укажу на дополнительную команду, созданную специально для проверки статуса развертывания:

```
$ kubectl rollout status deployment kubia
deployment kubia successfully rolled out
```

В соответствии с ней развертывание было успешно запущено, поэтому вы должны увидеть, что три реплики модуля работают. Давайте посмотрим:

```
$ kubectl get po
NAME                    READY  STATUS   RESTARTS  AGE
kubia-1506449474-otnnh  1/1    Running  0         14s
kubia-1506449474-vmn7s  1/1    Running  0         14s
kubia-1506449474-xis6m  1/1    Running  0         14s
```

Как развертывание создает наборы реплик, которые затем создают модули

Обратите внимание на имена этих модулей. Ранее, когда для создания модулей вы использовали контроллер репликации, их имена были составлены из имени контроллера плюс случайно сгенерированной строки (например, kubia-v1-m33mv). Три модуля, созданных развертыванием, содержат дополнительное числовое значение в середине их имен. Что оно значит?

Число соответствует хеш-значению шаблона модуля в развертывании и наборе реплик, управляющем этими модулями. Как мы уже говорили ранее, развертывание не управляет модулями напрямую. Вместо этого оно создает наборы реплик и отдает управление на их усмотрение, поэтому давайте посмотрим на набор реплик, созданный развертыванием:

```
$ kubectl get replicasets
NAME              DESIRED   CURRENT   AGE
kubia-1506449474  3         3         10s
```

Имя набора реплик также содержит хеш-значение шаблона модуля. Как вы увидите позже, развертывание создает несколько наборов реплик – по одному для каждой версии шаблона модуля. Использование хеш-значения шаблона модуля таким образом позволяет развертыванию всегда использовать один и тот же (возможно, существующий) набор реплик для данной версии шаблона модуля.

Доступ к модулям через службу

Теперь, когда три реплики, созданные этим наборов реплик, запущены, вы можете использовать службу, которую вы создали некоторое время назад, чтобы получить к ним доступ, потому что вы сделали так, чтобы метки новых модулей совпадали с селектором меток службы.

До этого момента вы, вероятно, не видели достаточно веской причины, по которой следует использовать развертывания в противовес контроллерам репликации. К счастью, создавать развертывание не сложнее, чем создавать контроллер репликации. Теперь вы начнете делать с этим развертыванием нечто осязаемое, что даст вам понять, почему развертывания имеют превосходство. Это станет ясно в ближайшие несколько мгновений, когда вы увидите, как обновление приложения через ресурс развертывания Deployment отличается от обновления через контроллер репликации.

9.3.2 Обновление с помощью развертывания

Раньше, когда вы запускали свои приложения с помощью контроллера репликации, вы должны были явным образом поручать Kubernetes делать обновление путем выполнения команды `kubectl rolling-update`. Вы даже должны были указать имя нового контроллера репликации, который должен был заменить старый. Kubernetes заменил все исходные модули на новые и в конце этого процесса удалил исходный контроллер репликации. Во время данного процесса вы в основном должны были оставаться неподалеку, держать терминал открытым и ожидать, когда `kubectl` закончит обновление.

Теперь сравните это с тем, как вы собираетесь обновить развертывание. Единственное, что вам нужно сделать, – это изменить шаблон модуля, определенный в ресурсе развертывания, и Kubernetes предпримет все необходимые шаги, чтобы привести фактическое состояние системы к тому, что определено в ресурсе. Подобно увеличению или уменьшению масштаба контроллера репликации или набора реплик, вам нужно всего лишь сослаться на новый тег образа в шаблоне модуля развертывания и предоставить его Kubernetes для преобразования системы, чтобы она стала соответствовать новому требуемому состоянию.

Стратегии развертывания

Способ достижения этого нового состояния определяется стратегией развертывания, конфигурируемой в самом развертывании. Стратегия по умолчанию – выполнять плавное обновление (эта стратегия называется `RollingUpdate`). Альтернативной ей является стратегия воссоздания `Recreate`, которая одним разом удаляет все старые модули, а затем создает новые, подобно изменению шаблона модуля контроллера репликации, и потом удаления всех модулей (мы говорили об этом в разделе 9.1.1).

Стратегия воссоздания `Recreate` приводит к удалению всех старых модулей перед созданием новых. Используйте эту стратегию, если приложение не поддерживает параллельного выполнения нескольких версий и требует полной остановки старой версии перед запуском новой. Эта стратегия включает в себя короткий период времени, когда ваше приложение станет полностью недоступным.

Стратегия `RollingUpdate`, с другой стороны, удаляет старые модули один за другим, одновременно добавляя новые, сохраняя приложение доступным на протяжении всего процесса и гарантируя отсутствие падения его способности обрабатывать запросы. Эта стратегия принята по умолчанию. Верхний и нижний пределы количества модулей выше или ниже требуемого количества реплик поддаются конфигурированию. Эту стратегию следует использовать только в том случае, если приложение может одновременно работать как со старой, так и с новой версией.

Способы изменения развертываний и других ресурсов

В ходе этой книги вы узнали несколько способов изменения существующего объекта. Давайте перечислим их все вместе, чтобы освежить вашу память.

Таблица 9.1. Модификация существующего ресурса в Kubernetes

Метод	Что он делает
`kubectl edit`	Открывает манифест объекта в редакторе, заданном по умолчанию. После внесения изменений, сохранения файла и выхода из редактора объект обновляется. Пример: `kubectl edit deployment kubia`
`kubectl patch`	Модифицирует отдельные свойства объекта. Пример: `kubectl patch deployment kubia -p '{"spec":{"template": {"spec": {"containers": [{"name": "nodejs", "image": "luksa/kubia:v2" }]}}}}'`
`kubectl apply`	Модифицирует объект, применяя значения свойств из полного файла YAML или JSON. Если объект, указанный в YAML/JSON, еще не существует, он будет создан. Файл должен содержать полное определение ресурса (он не может включать только поля, которые вы хотите обновить, как в случае с `kubectl patch`). Пример: `kubectl apply -f kubia-deployment -v2.yaml`
`kubectl replace`	Заменяет объект на новый объект из файла YAML/JSON. В отличие от команды `apply`, эта команда требует существования объекта; в противном случае выводится сообщение об ошибке. Пример: `kubectl replace -f kubia-deployment -v2.yaml`
`kubectl set image`	Меняет образ контейнера, определенный в модуле, шаблоне контроллера репликации, развертывании, наборе демонов, задании или наборе реплик. Пример: `kubectl set image deployment kubia nodejs=luksa/kubia:v2`

Что касается развертывания, все эти методы эквивалентны. Они изменяют спецификацию развертывания. Это изменение затем инициирует процесс раскрутки.

Замедление плавного обновления для демоцелей

В следующем далее упражнении вы будете использовать стратегию `RollingUpdate`, но вам нужно немного замедлить процесс обновления, что-

бы вы могли видеть, что обновление действительно выполняется плавным способом. Это можно сделать, установив атрибут `minReadySeconds` в развертывании. Мы объясним, что этот атрибут делает, в конце данной главы. Сейчас установите его на 10 секунд с помощью команды `kubectl patch`.

```
$ kubectl patch deployment kubia -p '{"spec": {"minReadySeconds": 10}}'
"kubia" patched
```

> **СОВЕТ.** Команда `kubectl patch` полезна для изменения одного свойства или ограниченного числа свойств ресурса без необходимости редактирования его определения в текстовом редакторе.

Команда `patch` используется для изменения спецификации развертывания. Она не приводит к обновлению модулей, так как шаблон модуля не был изменен. Изменение других свойств развертывания, таких как количество требуемых реплик или стратегии развертывания, также не инициирует выкладку, поскольку оно никак не влияет на существующие отдельные модули.

Инициирование плавного обновления

Если вы хотите отслеживать процесс обновления по мере его выполнения, сначала снова запустите цикл `curl` в другом терминале, чтобы увидеть, что происходит с запросами (не забудьте заменить IP-адрес фактическим внешним IP-адресом вашей службы):

```
$ while true; do curl http://130.211.109.222; done
```

Для того чтобы инициировать фактическую выкладку, вы измените образ, используемый в одном контейнере `luksa/kubia:v2`. Вместо того чтобы для изменения образа редактировать весь текст YAML объекта Deployment целиком либо использовать команду `patch`, вы будете применять команду `kubectl set image`, которая позволяет изменять образ любого ресурса, содержащего контейнер (контроллеры репликации, наборы реплик, развертывания и т. п.). Его можно использовать для изменения развертывания следующим образом:

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v2
deployment "kubia" image updated
```

При выполнении этой команды обновляется шаблон модуля развертывания `kubia`, поэтому образ, используемый в его контейнере `nodejs`, изменяется на `luksa/kubia:v2` (с `:v1` на `:v2`). Это показано на рис. 9.9.

Если вы запустите цикл `curl`, то увидите, что запросы сначала попадают только в модули `v1`, а потом все большее и большее их количество попадает в модули `v2` до тех пор, пока, наконец, все они не будут попадать только в оставшиеся модули версии `v2` , после того как все модули `v1` будут удалены. Во многом это работает так же, как скользящее обновление, выполняемое агентом `kubectl`.

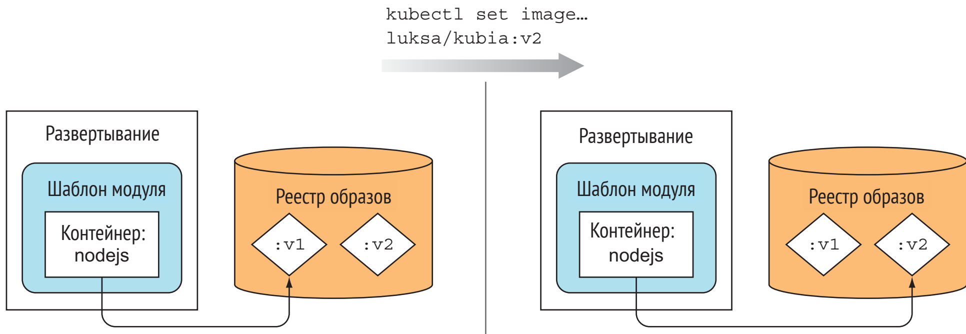


Рис. 9.9. Обновление шаблона модуля развертывания, заставляем его указывать на новый образ

Великолепие развертываний

Давайте подумаем о том, что произошло. Изменив шаблон модуля в ресурсе развертывания, вы обновили приложение до более новой версии – изменив одно-единственное поле!

Контроллеры, работающие в составе плоскости управления Kubernetes, затем выполнили обновление. Процесс не был выполнен клиентом `kubectl`, как это было при использовании `kubectl rolling-update`. Не знаю, как вы, но я думаю, что это проще, чем выполнять специальную команду, сообщающую системе Kubernetes, что делать, а потом ждать, пока этот процесс будет завершен.

> **ПРИМЕЧАНИЕ.** Имейте в виду, что если шаблон модуля в развертывании ссылается на ConfigMap (или Secret), то изменение словаря конфигурации не будет инициировать обновление. Один из способов инициировать обновление, когда вам нужно изменить приложение, – это создать новый словарь конфигурации и изменить шаблон модуля, чтобы он ссылался на новый словарь конфигурации.

События, произошедшие под поверхностью развертывания во время обновления, аналогичны событиям, произошедшим во время скользящего обновления `kubectl rolling-update`. Был создан дополнительный набор реплик, масштаб которого затем медленно увеличивался, в то время как масштаб предыдущего набора реплик уменьшался до нуля (начальное и конечное состояния показаны на рис. 9.10).

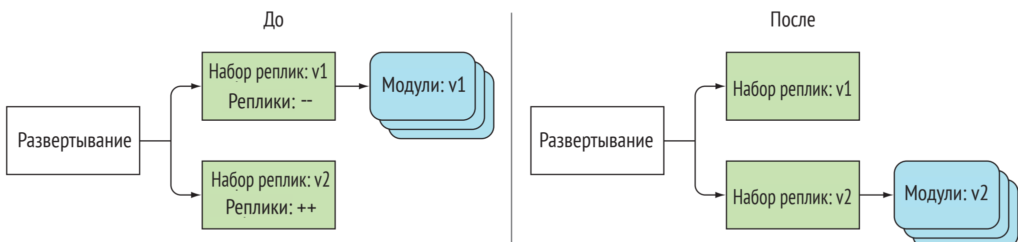


Рис. 9.10. Развертывания в начале и конце скользящего обновления

Вы по-прежнему можете увидеть старый набор реплик рядом с новым, если выведете их список:

```
$ kubectl get rs
NAME               DESIRED   CURRENT   AGE
kubia-1506449474   0         0         24m
kubia-1581357123   3         3         23m
```

Аналогично контроллеру репликации, все новые модули теперь управляются новым набором реплик. В отличие от того, как это было раньше, старый набор реплик по-прежнему существует, в то время как старый контроллер репликации был удален в конце процесса скользящего обновления. Вскоре вы увидите, какова цель этого неактивного набора реплик.

Однако вас не должны интересовать наборы реплик, потому что вы не создаете их непосредственно. Вы создали ресурс развертывания и оперируете только на нем; лежащие в основе наборы реплик являются деталью реализации. Согласитесь, управлять одним объектом развертывания намного проще, чем работать с несколькими контроллерами репликации и заниматься их отслеживанием.

Хотя эта разница может быть не столь очевидной, когда все складывается хорошо с выкладкой, она становится гораздо очевиднее, когда вы сталкиваетесь с проблемой во время процесса выкладки. Давайте смоделируем одну такую проблему прямо сейчас.

9.3.3 Откат развертывания

В настоящее время вы используете версию `v2` вашего образа, поэтому вам нужно сначала подготовить версию 3.

Создание версии 3 приложения

В версии 3 вы введете ошибку, которая заставит ваше приложение корректно обрабатывать только первые четыре запроса. Все запросы начиная с пятого будут возвращать внутреннюю ошибку сервера (код состояния HTTP 500). Вы будете симулировать это, добавив инструкцию `if` в начало функции-обработчика. В следующем списке показан новый программный код, а все необходимые изменения выделены жирным шрифтом.

Листинг 9.8. Версия 3 нашего приложения (нарушенная версия): v3/app.js

```
const http = require('http');
const os = require('os');

var requestCount = 0;

console.log("Kubia server starting...");

var handler = function(request, response) {
```

```
  console.log("Received request from " + request.connection.remoteAddress);
  if (++requestCount >= 5) {
    response.writeHead(500);
    response.end("Some internal error has occurred! This is pod " +
      os.hostname() + "\n");
    return;
  }
  response.writeHead(200);
  response.end("This is v3 running in pod " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

Как видим, на пятом и всех последующих запросах этот программный код возвращает ошибку 500 с сообщением «Some internal error has occurred...» (Произошла какая-то внутренняя ошибка...).

Развертывание версии 3

Я сделал версию `v3` образа доступной как `luksa/kubia: v3`. Вы развернете эту новую версию, снова поменяв образ в спецификации развертывания:

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v3
deployment "kubia" image updated
```

Вы можете следить за ходом раскрутки с помощью команды `kubectl rollout status`:

```
$ kubectl rollout status deployment kubia
Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for rollout to finish: 1 old replicas are pending termination...
deployment "kubia" successfully rolled out
```

Новая версия теперь готова. Как показано в следующем ниже листинге, после нескольких запросов веб-клиенты начинают получать ошибки.

Листинг 9.9. Попадание в нарушенную версию 3

```
$ while true; do curl http://130.211.109.222; done
This is v3 running in pod kubia-1914148340-lalmx
This is v3 running in pod kubia-1914148340-bz35w
This is v3 running in pod kubia-1914148340-w0voh
...
This is v3 running in pod kubia-1914148340-w0voh
Some internal error has occurred! This is pod kubia-1914148340-bz35w
This is v3 running in pod kubia-1914148340-w0voh
Some internal error has occurred! This is pod kubia-1914148340-lalmx
```

```
This is v3 running in pod kubia-1914148340-w0voh
Some internal error has occurred! This is pod kubia-1914148340-lalmx
Some internal error has occurred! This is pod kubia-1914148340-bz35w
Some internal error has occurred! This is pod kubia-1914148340-w0voh
```

Отмена раскрутки

Пользователи не должны получать внутренние ошибки сервера, поэтому необходимо быстро что-то с этим сделать. В разделе 9.3.6 вы увидите, как автоматически блокировать плохие раскрутки, но пока давайте посмотрим, что вы можете сделать с вашей плохой раскруткой вручную. К счастью, развертывания упрощают откат к ранее развернутой версии, сообщая Kubernetes отменить последнюю раскрутку развертывания:

```
$ kubectl rollout undo deployment kubia
deployment "kubia" rolled back
```

Эта команда возвращает развертывание к предыдущей редакции.

> **СОВЕТ.** Команда `undo` также может использоваться в ходе самого процесса раскрутки, чтобы практически прерывать раскрутку. Модули, уже созданные во время процесса раскрутки, удаляются и снова заменяются на старые.

Вывод истории раскрутки

Откат раскрутки возможен, поскольку в объектах Deployment хранится история ревизий. Как вы увидите позже, эта история хранится в наборах репликаций, созданных развертыванием. Когда выкладка завершается, старые наборы репликаций не удаляются, и это позволяет выполнять откат к любой ревизии, а не только к предыдущей. Историю ревизий можно отобразить с помощью команды `kubectl rollout history`:

```
$ kubectl rollout history deployment kubia
deployments «kubia»:
REVISION CHANGE-CAUSE
2 kubectl set image deployment kubia nodejs=luksa/kubia:v2
3 kubectl set image deployment kubia nodejs=luksa/kubia:v3
```

Помните параметр командной строки `--record`, который вы использовали при создании развертывания? Без него столбец `CHANGE-CAUSE` в истории ревизий был бы пустым, что значительно затруднило бы выяснение того, что стоит за каждой ревизией.

Откат к определенной версии развертывания

Вы можете откатиться к определенной ревизии, указав ревизию в команде `undo`. Например, если вы хотите выполнить откат к первой версии, выполните следующую ниже команду:

```
$ kubectl rollout undo deployment kubia --to-revision=1
```

Помните, как остался неактивный набор реплик, когда вы модифицировали развертывание в первый раз? Этот набор реплик представляет первую ревизию развертывания. Как показано на рис. 9.11, все наборы реплик, созданные развертыванием, содержат полную историю ревизий. Каждый набор реплик хранит полную информацию о развертывании в этой конкретной ревизии, поэтому не следует удалять ее вручную. Если вы это сделаете, то потеряете эту конкретную ревизию из истории развертывания, что не позволит вам вернуться к ней в дальнейшем.

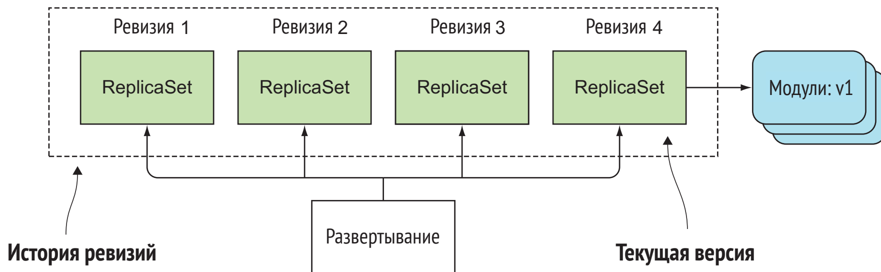


Рис. 9.11. Наборы реплик развертывания также выступают в качестве его истории ревизий

Однако наличие старых наборов реплик, которые загромождают ваш список реплик, – это не самый идеальный случай, поэтому длина истории ревизий ограничивается свойством `revisionHistoryLimit` на ресурсе развертывания Deployment. По умолчанию его значение равно двум, поэтому обычно в истории отображаются только текущая и предыдущая ревизии (и сохраняются только текущая и предыдущая наборы реплик). Старшие наборы реплик удаляются автоматически.

> **ПРИМЕЧАНИЕ.** Версия развертываний `extensions/v1beta1` не имеет заданного по умолчанию лимита `revisionHistoryLimit`, в то время как в версии `apps/v1beta2` он по умолчанию равен 10.

9.3.4 Управление скоростью выкладки

Когда вы выполняли раскрутку до версии 3 и отслеживали ее ход с помощью команды `kubectl rollout status`, вы увидели, что сначала был создан новый модуль, а когда он стал доступен, один из старых модулей был удален, и еще один новый модуль был создан. Это продолжалось до тех пор, пока не осталось старых модулей. Способ создания новых и удаления старых модулей настраивается с помощью двух дополнительных свойств стратегии скользящего обновления.

Знакомство со свойствами maxSurge и maxUnavailable стратегии скользящего обновления

На количество модулей, заменяемых одновременно во время скользящего обновления развертывания, влияют два свойства. Это `maxSurge` и `maxUnavailable`. Они могут быть заданы в рамках вложенного подсвойства `rollingUpdate` атрибута `strategy` развертывания, как показано в следующем ниже листинге.

Листинг 9.10. Задание параметров для стратегии rollingUpdate

```
spec:
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```

Характер выполняемых этими свойствами действий объясняется в табл. 9.2.

Таблица 9.2. Свойства для настройки скорости обновления

Свойство	Что оно делает
`maxSurge`	Определяет, скольким экземплярам модуля вы позволяете существовать выше требуемого количества реплик, настроенного на развертывании. По умолчанию используется значение 25%, поэтому количество экземпляров модуля может быть не более 25%. Если требуемое количество реплик равно четырем, то во время обновления никогда не будет одновременно запущено более пяти экземпляров модуля. При преобразовании процента в абсолютное число это число округляется вверх. Вместо процента это значение может быть абсолютным (например, можно разрешить один или два дополнительных модуля)
`maxUnavailable`	Определяет, сколько экземпляров модуля может быть недоступно относительно требуемого количества реплик во время обновления. Оно также по умолчанию равно 25%, поэтому количество доступных экземпляров модуля никогда не должно опускаться ниже 75% от требуемого количества реплик. Здесь при преобразовании процента в абсолютное число это число округляется вниз. Если требуемое число реплик равно четырем и процент составляет 25%, то только один модуль может быть недоступен. В течение всей раскрутки всегда будет доступно для обслуживания запросов, по крайней мере, три экземпляра модуля. Как и в случае с `maxSurge`, вместо процента можно также указать абсолютное значение

Поскольку требуемое количество реплик в вашем случае составляло три и оба эти свойства по умолчанию равны 25%, свойство `maxSurge` позволило ко-

личеству всех модулей достичь четырех, а свойство maxUnavailable запретило иметь какие-либо недоступные модули (другими словами, три модуля должны были быть доступны в любое время). Это показано на рис. 9.12.

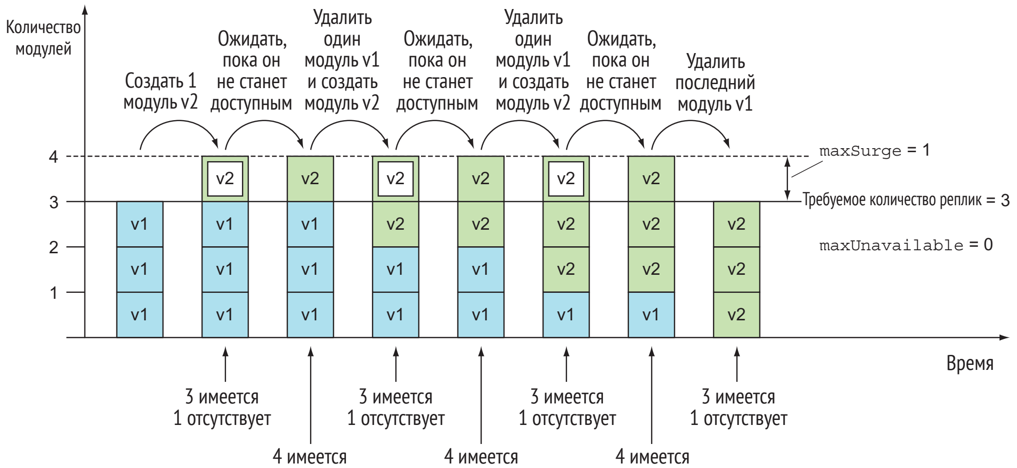


Рис. 9.12. Плавное обновление при развертывании с тремя репликами и значениями maxSurge и maxUnavailable по умолчанию

Свойство maxUnavailable подробнее

В версии extensions/v1beta1 развертываний используются другие значения по умолчанию – в ней и maxSurge, и maxUnavailable вместо 25% получают значение 1. В случае трех реплик свойство maxSurge такое же, как и раньше, но свойство maxUnavailable отличается (1 вместо 0). Это заставляет развертывать процесс раскрутки немного по-другому, как показано на рис. 9.13.

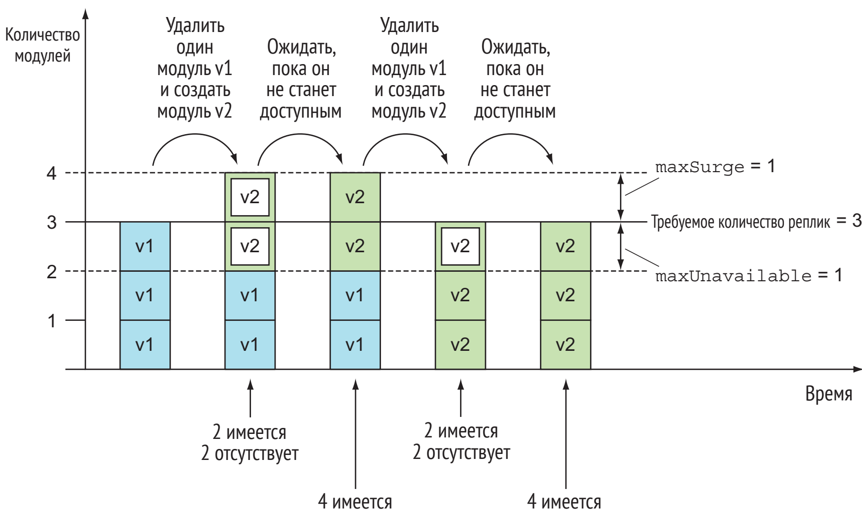


Рис. 9.13. Плавное обновление при развертывании с maxSurge=1 и maxUnavailable=1

В этом случае одна реплика может быть недоступна, поэтому если требуемое количество реплик равно трем, то должны быть доступны только две из них. Вот почему процесс раскрутки немедленно удаляет один модуль и создает два новых. Это гарантирует, что два модуля доступны и что максимальное количество модулей не превышено (в данном случае максимум четыре – три плюс один от `maxSurge`). Как только два новых модуля становятся доступными, два оставшихся старых модуля удаляются.

Это немного трудно понять, тем более что свойство `maxUnavailable` заставляет вас полагать, что это максимальное количество недоступных модулей, которые разрешены. Если вы рассмотрите предыдущий рисунок внимательно, то увидите два недоступных модуля во втором столбце, несмотря на то что свойство `maxUnavailable` имеет значение 1.

Важно иметь в виду, что свойство `maxUnavailable` относится к требуемому количеству реплик. Если количество реплик равно трем, а значение свойства `maxUnavailable` равно одному, то это означает, что в процессе обновления всегда должно быть доступно, по крайней мере, два (3 минус 1) модуля, а число недоступных модулей может превышать один.

9.3.5 Приостановка процесса выкладки

После неудачного опыта с версией 3 вашего приложения представьте, что вы теперь исправили ошибку и отправили версию 4 вашего образа. У вас есть некоторые опасения относительно ее раскрутки по всем вашим модулям, как вы это делали раньше. Все, чего вы хотите, – это запустить один модуль `v4` рядом с существующими модулями `v2` и посмотреть, как он поведет себя только с долей всех ваших пользователей. Затем, как только вы будете уверены, что все в порядке, вы сможете заменить все старые модули на новые.

Этого можно добиться, запустив дополнительный модуль непосредственно или с помощью дополнительного развертывания, контроллера репликации и набора реплик, но у вас есть другой вариант, заключенный в самом развертывании. Развертывание можно приостановить во время процесса раскрутки. Это позволяет проверять, что с новой версией все в порядке, перед тем как приступать к остальной части раскрутки.

Приостановка раскрутки

Я подготовил образ `v4`, поэтому продолжайте и инициируйте раскрутку, заменив образ на `luksa/kubia: v4`, но затем немедленно (в течение нескольких секунд) приостановите раскрутку:

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v4
deployment "kubia" image updated

$ kubectl rollout pause deployment kubia
deployment "kubia" paused
```

Должен быть создан один новый модуль, но все исходные модули тоже должны быть запущены. Как только новый модуль будет запущен, часть всех запросов в службу будет перенаправлена на новый модуль. Таким образом, вы практически запускаете канареечный выпуск приложения. Канареечный выпуск – это метод минимизации риска выкладки плохой версии приложения и его влияния на всех пользователей. Вместо того чтобы выкладывать новую версию для всех, вы заменяете только один или небольшое количество старых модулей на новые. Благодаря этому только небольшое количество пользователей изначально попадет в новую версию. Затем можно проверить, работает новая версия нормально или нет, а после этого либо продолжить выкладывать по всем оставшимся модулям, либо вернуться к предыдущей версии.

Возобновление выкладки

В вашем случае, приостановив процесс выкладки, только небольшая часть клиентских запросов попадет в ваш модуль v4, в то время как большинство по-прежнему попадет в модули v3. Если вы уверены, что новая версия работает должным образом, выкладку можно возобновить и заменить все старые модули на новые:

```
$ kubectl rollout resume deployment kubia
deployment "kubia" resumed
```

Очевидно, что приостановка развертывания в определенный момент процесса выкладки – это не то, что вы хотите сделать. В будущем новая стратегия обновления, возможно, сделает это автоматически, но в настоящее время правильный способ выполнения канареечного выпуска – использовать два разных развертывания и соответствующее их масштабирование.

Использование функционала паузы для предотвращения раскруток

Приостановка развертывания также может использоваться для того, чтобы не давать обновлениям в развертывании запускать процесс раскрутки, что позволяет вам вносить множественные изменения в развертывание и запускать раскрутку только после внесения всех необходимых изменений. После того как изменения вступят в силу, вы возобновляете развертывание, и процесс раскрутки запустится.

> **ПРИМЕЧАНИЕ.** Если развертывание приостановлено, то команда undo его не отменит до тех пор, пока вы не возобновите развертывание.

9.3.6 Блокировка раскруток плохих версий

Прежде чем завершить эту главу, необходимо обсудить еще одно свойство ресурса развертывания Deployment. Помните свойство minReadySeconds, установленное для развертывания в начале раздела 9.3.2? Вы использовали его для того, чтобы замедлить раскрутку и увидеть, что оно действительно

выполняет скользящее обновление, а не заменяет все модули сразу. Главной функцией свойства `minReadySeconds` является предотвращение развертывания неисправных версий, а не замедление развертывания ради удовольствия.

Применимость minReadySeconds

Свойство `minReadySeconds` задает, как долго вновь созданный модуль должен быть в готовности, прежде чем модуль будет считаться доступным. До тех пор, пока модуль не будет доступен, процесс раскрутки не будет продолжаться (помните свойство `maxUnavailable`?). Модуль находится в готовности, когда проверки готовности всех его контейнеров возвращают успех. Если новый модуль не функционирует должным образом и его проверка готовности начинает не срабатывать до того, как пройдут секунды `minReadySeconds`, раскрутка новой версии будет практически заблокирована.

Вы использовали это свойство, чтобы замедлить процесс раскрутки, заставив Kubernetes ждать 10 секунд, после того как модуль был готов, прежде чем продолжить раскрутку. Обычно свойству `minReadySeconds` назначают гораздо большее значение, чтобы убедиться, что модули продолжают сообщать о своей готовности, после того как они уже начали получать фактический трафик.

Хотя вы, безусловно, должны протестировать свои модули как в тестовой, так и в промежуточной (стейдж) среде до того, как развертывать их в рабочее окружение, использование свойства `minReadySeconds` похоже на подушку безопасности, которая спасает ваше приложение от большого хаоса, после того как вы уже позволили дефектной версии проскользнуть в производство.

С правильно сконфигурированной проверкой готовности и правильной настройкой свойства `minReadySeconds` система Kubernetes помешала бы нам ранее развернуть дефектную версию `v3`. Давайте покажу, как.

Определение проверки готовности для предотвращения полной раскрутки версии v3

Вы собираетесь снова развернуть версию `v3`, но на этот раз у вас будет соответствующая проверка готовности, определенная на модуле. Ваше развертывание в настоящее время находится в версии `v4`, поэтому перед запуском снова откатите до версии `v2`, чтобы вы могли притвориться, что это первое обновление до `v3`. Если хотите, вы можете перейти прямо от `v4` к `v3`, но следующий ниже текст предполагает, что вы сначала вернулись к `v2`.

В отличие от того, как это было раньше, где вы обновляли только образ в шаблоне модуля, теперь же вы одновременно внесете проверку готовности контейнера. До этого, поскольку проверка готовности не была задана явно, контейнер и модуль всегда считались готовыми, даже если приложение не было действительно готово или возвращало ошибки. И система Kubernetes не могла никоим образом узнать, что приложение функционирует неправильно и не должно предоставляться клиентам.

Для того чтобы изменить образ и сразу внести проверку готовности, используйте команду `kubectl apply`. Как показано в следующем ниже листинге, для обновления развертывания вы будете использовать следующий ниже YAML (вы будете хранить его как `kubia-deployment-v3-with-readinesscheck.yaml`).

Листинг 9.11. Развертывание с помощью проверки готовности: kubia-deployment-v3-withreadinesscheck.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  minReadySeconds: 10          <- Вы задаете minReadySeconds равным 10
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0        <- Вы задаете maxUnavailable равным 0,
    type: RollingUpdate           чтобы развертывание заменяло модули
  template:                       один за другим
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
      - image: luksa/kubia:v3
      name: nodejs
      readinessProbe:
        periodSeconds: 1       <- Вы определяете проверку готовности, которая будет
                                  выполняться каждую секунду
        httpGet:
          path: /              <- Проверка готовности выполнит запрос
          port: 8080              HTTP GET к нашему контейнеру
```

Обновление развертывания с помощью команды kubectl apply

Чтобы обновить развертывание, на этот раз вы воспользуетесь командой `kubectl apply` следующим образом:

```
$ kubectl apply -f kubia-deployment-v3-with-readinesscheck.yaml
deployment "kubia" configured
```

Команда `apply` обновляет развертывание всем тем, что определено в файле YAML. Она не только обновляет образ, но и добавляет определение проверки готовности и все остальное, что вы добавили или изменили в YAML. Если новый YAML также содержит поле `replicas`, которое не соответствует количест-

ву реплик в существующем развертывании, операция `apply` промасштабирует развертывание, что, как правило, не является тем, что вы хотите.

> **СОВЕТ.** Для того чтобы при обновлении развертывания с помощью команды `kubectl apply` оставить требуемое количество реплик без изменений, не включайте поле `replicas` в YAML.

Выполнение команды `apply` запустит процесс обновления, за которым можно снова проследить с помощью команды `rollout status`:

```
$ kubectl rollout status deployment kubia
Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
```

Поскольку статус говорит о том, что был создан один новый модуль, ваша служба должна время от времени в него попадать, не так ли? Давайте посмотрим:

```
$ while true; do curl http://130.211.109.222; done
This is v2 running in pod kubia-1765119474-jvslk
This is v2 running in pod kubia-1765119474-jvslk
This is v2 running in pod kubia-1765119474-xk5g3
This is v2 running in pod kubia-1765119474-pmb26
This is v2 running in pod kubia-1765119474-pmb26
This is v2 running in pod kubia-1765119474-xk5g3
...
```

Нет, вы ни разу не попали в модуль `v3`. В чем причина? Существует ли он вообще? Выведем список модулей:

```
$ kubectl get po
NAME READY STATUS RESTARTS AGE
kubia-1163142519-7ws0i 0/1 Running 0 30s
kubia-1765119474-jvslk 1/1 Running 0 9m
kubia-1765119474-pmb26 1/1 Running 0 9m
kubia-1765119474-xk5g3 1/1 Running 0 8m
```

Ага! Вот в чем проблема (или, как вы скоро узнаете, ваша радость)! Модуль показан как неготовый, но я полагаю, что вы этого ожидали, верно? И что же случилось?

Проверка готовности предотвращает выкладку плохих версий

Как только ваш новый модуль запускается, проверка готовности начинает получать посещения каждую секунду (в секции `spec` модуля вы установили интервал проверки в одну секунду). После пятого запроса проверка готовности начала давать несработку, так как начиная с пятого запроса приложение возвращает код состояния HTTP 500.

В результате модуль удаляется в качестве конечной точки из службы (см. рис. 9.14). К тому времени, когда вы начинаете попадать в службу в цикле `curl`, модуль уже был помечен как неготовый. Это объясняет, почему вы ни разу не попали в новый модуль с помощью утилиты `curl`. И это именно то, что вы хотите, потому что вы не хотите, чтобы клиенты попадали в модуль, который не функционирует правильно.

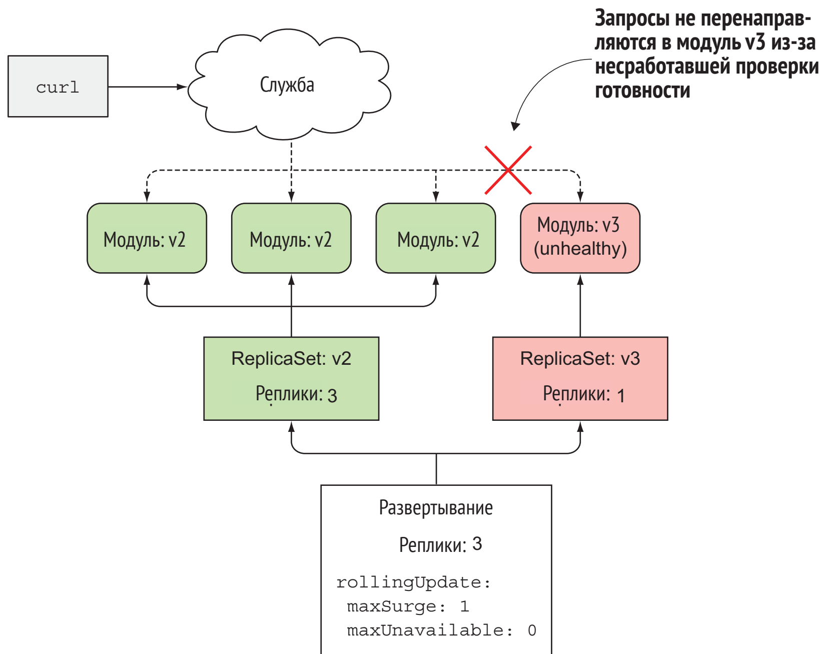


Рис. 9.14. Развертывание заблокировано несрабатывающей проверкой готовности в новом модуле

Но как насчет процесса выкладки? Команда `rollout status` показывает, что запущена всего одна новая реплика. К счастью, процесс раскрутки не будет продолжаться, потому что новый модуль никогда не станет доступным. Для того чтобы считаться доступным, он должен быть готов в течение, по крайней мере, 10 секунд. Пока он не станет доступен, процесс раскрутки не создаст новые модули, а также не удалит исходные модули, так как для свойства `maxUnavailable` задано значение 0.

Тот факт, что развертывание застряло, является хорошим признаком, потому что если бы оно продолжало заменять старые модули на новые, то вы бы в конечном итоге получили совершенно нерабочую службу, как вы это сделали, впервые развернув версию 3, когда вы не использовали проверку готовности. Но теперь, с проверкой готовности на своем месте, на ваших пользователей не было оказано практически никакого негативного влияния. Несколько пользователей, возможно, попало на внутреннюю ошибку сервера, но это не так страшно, как если бы развертывание заменило все модули неисправной версией 3.

СОВЕТ. Если вы определите только проверку готовности без правильной установки свойства `minReadySeconds`, то новые модули будут считаться доступными сразу же, когда первый вызов проверки готовности закончится успешно. Если вскоре после этого проверка готовности начинает не срабатывать, то плохая версия будет раскручена по всем модулям. Поэтому вы должны правильно настроить свойство `minReadySeconds`.

Настройка крайнего срока выкладки

По умолчанию, после того как развертывание не может выполнить никаких действий в течение 10 минут, оно считается неуспешным. Если вы воспользуетесь командой `kubectl describe deployment`, то увидите, что она отображает условие `ProgressDeadlineExceeded`, как показано в следующем ниже листинге.

Листинг 9.12. Просмотр условий развертывания с помощью команды kubectl describe

```
$ kubectl describe deploy kubia
Name: kubia
...
Conditions:
  Type          Status   Reason
  ----          ------   ------
  Available     True     MinimumReplicasAvailable
  Progressing   False    ProgressDeadlineExceeded
```

Развертывание заняло слишком много времени на достижение успешного результата

Время, через которое развертывание считается неуспешным, конфигурируется посредством свойства `progressDeadlineSeconds` в секции `spec` развертывания.

ПРИМЕЧАНИЕ. В версии `extensions/v1beta1` развертывания крайний срок не установлен.

Прерывание плохой раскрутки

Поскольку раскрутка никогда не будет продолжаться, единственное, что нужно сейчас сделать, – это прервать раскрутку, отменив ее:

```
$ kubectl rollout undo deployment kubia
deployment "kubia" rolled back
```

ПРИМЕЧАНИЕ. В будущих версиях раскрутка будет прерываться автоматически, когда время, указанное в свойстве `progressDeadlineSeconds`, будет превышено.

9.4 Резюме

В этой главе показано, как упростить свою жизнь с помощью декларативного подхода к развертыванию и обновлению приложений в Kubernetes. Теперь, когда вы прочитали данную главу, вы должны знать, как:

- выполнять скользящее обновление модулей, управляемых с помощью контроллера репликации;
- создавать развертывания вместо более низкоуровневых контроллеров репликации или наборов реплик;
- обновлять свои модули путем редактирования шаблона модуля в спецификации развертывания;
- откатывать развертывания к предыдущей ревизии либо к любой предыдущей ревизии, остающейся в истории ревизий;
- прерывать развертывания на полпути;
- приостанавливать развертывание, чтобы проинспектировать, как работает один экземпляр новой версии, прежде чем разрешить другим экземплярам модуля заменить старые;
- управлять скоростью скользящего обновления посредством свойств `maxSurge` и `maxUnavailable`;
- применять свойство `minReadySeconds` и проверку готовности, чтобы автоматически блокировать раскрутку дефектной версии.

В дополнение к этим специфическим для развертывания задачам вы также узнали, как:

- использовать три символа тире в качестве разделителя для описания нескольких ресурсов в одном файле YAML;
- включать подробное журналирование в `kubectl`, чтобы точно видеть, что этот агент делает за кулисами.

Теперь вы знаете, как развертывать и управлять наборами модулей, созданных из одного шаблона модуля и, следовательно, делящих между собой одно и то же постоянное хранилище. Вы даже знаете, как обновлять их декларативно. Но как насчет управления набором модулей, где каждый экземпляр должен использовать свое собственное постоянное хранилище? Мы этого еще не рассматривали. И это тема нашей следующей главы.

Глава 10

Ресурсы StatefulSet: развертывание реплицируемых приложений с внутренним состоянием

Эта глава посвящена:

- развертыванию кластерных приложений с внутренним состоянием;
- предоставлению отдельного хранилища для каждого экземпляра реплицируемого модуля;
- гарантированию стабильного имени и хостнейма для реплик модуля;
- запуску и останову реплик модулей в предсказуемом порядке;
- обнаружению связей с помощью записей SRV в DNS.

Теперь вы знаете, как запускать одиночные и реплицированные модули без внутреннего состояния и даже модули с внутренним состоянием, использующие постоянное хранилище. Вы можете запускать несколько реплицированных экземпляров модуля веб-сервера и один экземпляр модуля базы данных, использующий постоянное хранилище, предоставленное либо через простые тома модуля, либо через тома постоянного хранения PersistentVolume, связанные с заявкой на получение тома постоянного хранения PersistentVolumeClaim. Но можно ли использовать набор реплик ReplicaSet для репликации модуля базы данных?

10.1 Репликация модулей с внутренним состоянием

Наборы ReplicaSet создают множество реплик модуля из одного шаблона модуля. Эти реплики не отличаются друг от друга, кроме как по имени и IP-ад-

ресу. Если шаблон модуля содержит том, который относится к конкретной заявке на получение тома постоянного хранения (PersistentVolumeClaim), то все реплики набора реплик ReplicaSet будут использовать ту же заявку и, следовательно, тот же том постоянного хранения PersistentVolume, связанный с заявкой (см. рис. 10.1).

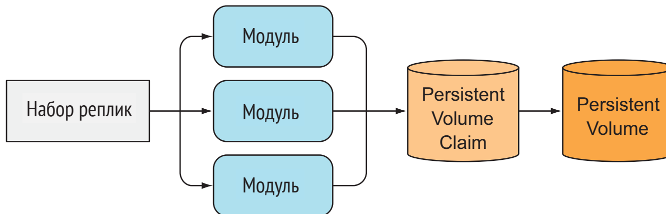


Рис. 10.1. Все модули из одного набора ReplicaSet всегда используют ту же заявку PersistentVolumeClaim и том PersistentVolume

Поскольку ссылка на заявку находится в шаблоне модуля, который используется для штамповки множества реплик модуля, сделать так, чтобы каждая реплика использовала отдельную заявку PersistentVolumeClaim, не получится. Нельзя использовать наборы реплик для управления распределенным хранилищем данных, где каждый экземпляр нуждается в отдельном хранилище, – по крайней мере, не с помощью одного набора реплик ReplicaSet. Честно говоря, ни один из объектов API, которые вы видели до сих пор, не делает возможным управление таким хранилищем данных. Вам нужно что-то еще.

10.1.1 Запуск множества реплик с отдельным хранилищем для каждой

Как запустить несколько реплик модуля и заставить каждый модуль использовать собственный том хранилища? Наборы ReplicaSet создают точные копии (реплики) модуля, поэтому их нельзя использовать для этих типов модулей. Что же тогда можно использовать?

Создание модулей вручную

Вы могли бы создавать модули вручную и делать так, чтобы каждый из них использовал свою собственную заявку PersistentVolumeClaim, но поскольку никакой набор реплик ими не занимается, вам пришлось бы управлять ими вручную и воссоздавать их, когда они исчезают (как в случае аварийного прекращения работы узла). Поэтому это не жизнеспособный вариант.

Использование одного набора реплик в расчете на экземпляр модуля

Вместо того чтобы создавать модули непосредственно, вы можете создать несколько наборов ReplicaSet – по одному для каждого модуля, в котором требуемое количество реплик каждого набора реплик равняется одному и каждый шаблон модуля в наборе реплик ссылается на выделенную заявку PersistentVolumeClaim (PVC) (как показано на рис. 10.2).

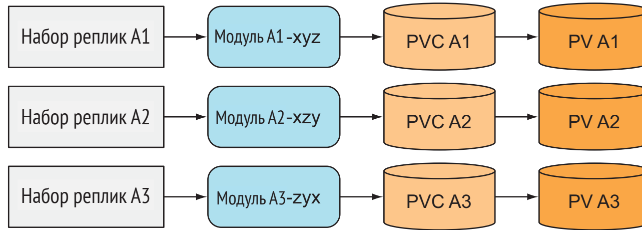


Рис. 10.2. Использование одного набора реплик для каждого экземпляра модуля

Хотя этот вариант берет на себя автоматическое переназначение в случае аварийного прекращения работы узла либо случайных удалений модуля, он гораздо более громоздкий, по сравнению с одним набором реплик. Например, подумайте о том, как промасштабировать модули в этом случае. Вы не сможете поменять нужное количество реплик – вместо этого вам придется создавать дополнительные наборы реплик.

Использование множества наборов реплик поэтому не является самым лучшим решением. Но, может быть, вы могли бы использовать один набор реплик ReplicaSet и давать каждому экземпляру модуля поддерживать свое постоянное состояние, даже если все они используют одинаковый том хранения данных?

Использование множества каталогов в одном томе

Трюк состоит в том, чтобы все модули использовали тот же самый постоянный том PersistentVolume, но при этом имели отдельный файловый каталог внутри этого тома для каждого модуля (это показано на рис. 10.3).

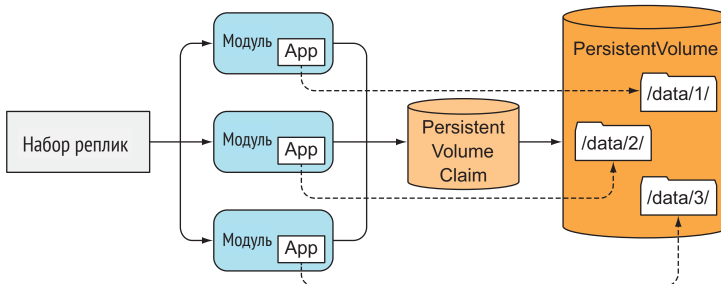


Рис. 10.3. Обход проблемы общего хранилища с предоставлением каждому модулю возможности использовать другой файловый каталог

Раз вы не можете сконфигурировать реплики модуля так, чтобы они отличались от единого шаблона модуля, вы не сможете сообщить каждому экземпляру, какой каталог он должен использовать, однако вы можете заставить каждый экземпляр автоматически выбирать (и, возможно, также создавать) каталог данных, который в то же самое время не используется ни одним другим экземпляром. Это решение требует координации между экземплярами, и его не просто реализовать на практике правильно. Оно также делает общий том хранения узким местом.

10.1.2 Обеспечение стабильной долговременной идентификации для каждого модуля

В дополнение к хранилищу некоторые кластеризованные приложения также требуют, чтобы каждый экземпляр имел долгосрочную стабильную возможность идентифицировать конкретный экземпляр (идентификацию). Модули могут время от времени уничтожаться и заменяться новыми. Когда набор реплик сменяет модуль, новый модуль – это совершенно новый модуль с новым хостнеймом и IP-адресом, несмотря на то что данные в его томе хранения могут быть данными уничтоженного модуля. Для некоторых приложений запуск с данными старого экземпляра, но с совершенно новой сетевой идентичностью может вызывать проблемы.

Почему некоторые приложения требуют стабильной сетевой идентификации? Это требование довольно распространено в распределенных приложениях с внутренним состоянием. Некоторые приложения требуют от администратора иметь список всех других членов кластера и их IP-адреса (или хостнейма) в файле конфигурации каждого члена. Но в Kubernetes всякий раз, когда модуль переназначается, новый модуль получает как новый хостнейм, так и новый IP-адрес, поэтому весь прикладной кластер должен подвергаться переконфигурированию всякий раз, когда один из его членов переназначается.

Использование выделенной службы для каждого экземпляра модуля

Чтобы обойти эту проблему, можно создать стабильный сетевой адрес для членов кластера, организовав для каждого отдельного члена выделенную службу Kubernetes. Поскольку IP-адреса служб стабильны, вы можете указать на каждого участника посредством его IP-адреса службы (а не IP-адреса модуля) в конфигурации.

Это похоже на создание набора реплик для каждого члена, чтобы предоставить им индивидуальное хранилище, как описано выше. Объединение этих двух методов приводит к структуре, показанной на рис. 10.4 (также показана дополнительная служба, охватывающая всех членов кластера, потому что обычно такая служба требуется для клиентов кластера).

Подобное решение не только уродливое, но оно по-прежнему не решает всех вопросов. Отдельные модули не могут знать, через какую службу обеспечивается к ним доступ (и, следовательно, не могут знать свой стабильный IP-адрес), поэтому, используя этот IP-адрес, они не могут саморегистрироваться в других модулях.

К счастью, Kubernetes избавляет нас от таких сложных решений. Правильный и простой способ запуска этих специальных типов приложений в Kubernetes осуществляется посредством набора StatefulSet, то есть набора модулей с внутренним состоянием.

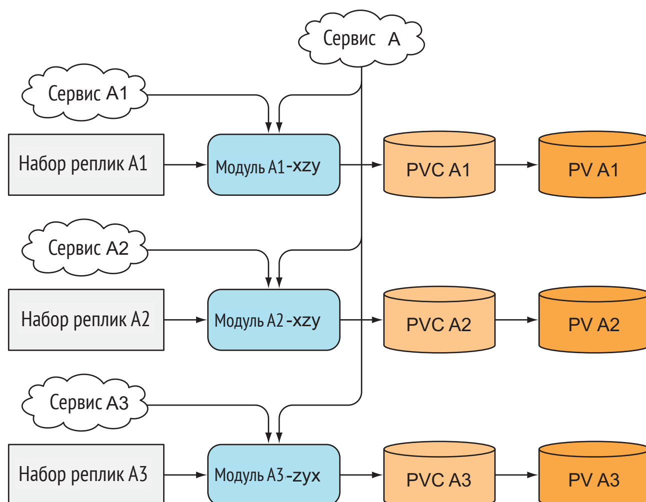


Рис. 10.4. Использование одной службы и набора реплик для каждого модуля для предоставления стабильного сетевого адреса и отдельного тома для каждого модуля

10.2 Набор модулей с внутренним состоянием

Вместо использования реплик для запуска этих типов модулей создается набор StatefulSet, специально предназначенный для приложений, в которых экземпляры приложения должны рассматриваться как невзаимозаменяемые сущности, причем каждый из них должен иметь стабильное имя и состояние.

10.2.1 Сопоставление наборов модулей с внутренним состоянием и наборов реплик

Чтобы понять назначение наборов StatefulSet, лучше сравнить их с наборами реплик ReplicaSet или контроллерами репликации ReplicationController. Но сначала следует дать им объяснение, приведя небольшую аналогию, которая широко используется в этой области.

Модули с внутренним состоянием в аналогии с домашними животными и крупным рогатым скотом

Возможно, вы уже слышали об аналогии «Домашние животные vs. крупный рогатый скот» (Pets vs. Cattle). Если нет, давайте объясню. Мы можем рассматривать наши приложения как домашних животных или как стадо.

ПРИМЕЧАНИЕ. Наборы модулей с внутренним состоянием StatefulSet изначально назывались наборами PetSet, то есть наборами-питомцами. Это название происходит от объясненной здесь аналогии про домашних животных против стада.

Мы склонны рассматривать наши экземпляры приложений как домашних животных, где мы каждому экземпляру даем имя и индивидуально заботимся о каждом экземпляре. Но обычно лучше рассматривать экземпляры как стадо и не обращать особого внимания на каждый отдельный экземпляр. Это облегчает замену нездоровых экземпляров, совершенно не задумываясь об этом, подобно тому, как фермер заменяет нездоровую скотину.

Экземпляры приложения без внутреннего состояния, например, ведут себя как поголовье скота. Не имеет значения, умрет экземпляр или нет, – вы можете создать новый экземпляр, и люди не заметят разницы.

С другой стороны, в приложениях с внутренним состоянием экземпляр приложения больше похож на домашнего питомца. Когда питомец умирает, вы не можете купить нового и ожидать, что люди этого не заметят. Для того чтобы заменить потерянного питомца, вам нужно найти нового, который выглядит и ведет себя точно так же, как старый. В случае с приложениями это означает, что новый экземпляр должен иметь то же состояние и идентичность, что и старый.

Сопоставление наборов StatefulSet с наборами ReplicaSet и контроллерами репликации ReplicationController

Реплики модуля, управляемые набором ReplicaSet или контроллером репликации ReplicationController, очень похожи на стадо. Поскольку они в основном не хранят состояние, их в любое время можно поменять на совершенно новую реплику модуля. Модули с внутренним состоянием требуют другого подхода. Когда экземпляр модуля с внутренним состоянием умирает (или узел, на котором он работает, аварийно завершает работу), экземпляр модуля должен быть восстановлен на другом узле, но новый экземпляр должен получить то же самое имя, сетевую идентичность и состояние, что и заменяемый. Это то, что происходит, когда модули управляются посредством наборов с внутренним состоянием StatefulSet.

Набор StatefulSet гарантирует, что модули переназначаются таким образом, что они удерживают свою идентичность и состояние. Это также позволяет легко масштабировать количество питомцев вверх и вниз. У набора модулей с внутренним состоянием StatefulSet, как и у набора реплик ReplicaSet, есть поле требуемого количества реплик, которое определяет, сколько питомцев вы хотите запустить в это время. Аналогично наборам ReplicaSet, модули создаются из шаблона модуля, задаваемого в рамках набора StatefulSet (помните аналогию с формой для печенья?). Но, в отличие от модулей, создаваемых наборами ReplicaSet, модули, создаваемые набором StatefulSet, не являются точными копиями друг друга. Каждый может иметь свой собственный набор томов – другими словами, хранилище (и, следовательно, постоянное состояние), – который отличает его от своих соседей. Модули-питомцы также имеют предсказуемую (и стабильную) идентичность, вместо того чтобы каждый новый экземпляр модуля получал совершенно случайную.

10.2.2 Обеспечение стабильной сетевой идентичности

Каждому модулю, создаваемому набором StatefulSet, присваивается порядковый индекс (с отсчетом от нуля), который затем используется, чтобы произвести имя и хостнейм модуля, и закрепить за этим модулем надежное хранилище. Следовательно, имена модулей предсказуемы, поскольку имя каждого модуля является производным от имени набора StatefulSet и порядкового индекса экземпляра. Вместо того чтобы иметь случайные имена, эти модули организованы в порядке, как показано на следующем ниже рис. 10.5.

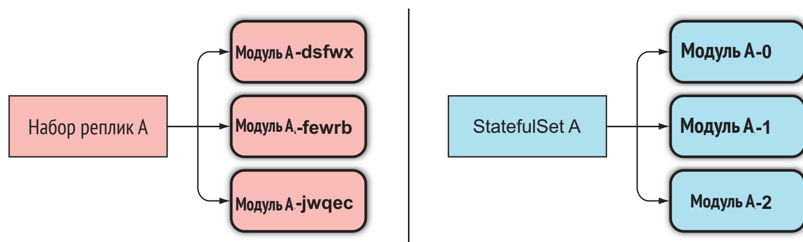


Рис. 10.5. Модули, создаваемые набором StatefulSet, имеют предсказуемые имена (и хостнеймы), в отличие от создаваемых набором ReplicaSet

Знакомство с управляющей службой

Однако дело вовсе не заканчивается тем, что такие модули имеют предсказуемое имя и хостнейм. В отличие от обычных модулей, модулям с внутренним состоянием иногда нужно быть адресуемыми по их хостнейму, в то время как модулям без внутреннего состояния, как правило, этого не нужно. Ведь каждый модуль без внутреннего состояния похож на любой другой. Когда вам нужен один, вы выбираете любой из них. Но в случае с модулями с внутренним состоянием вы обычно хотите работать на конкретном модуле из группы, потому что они отличаются друг от друга (они хранят разное состояние, например).

По этой причине набор StatefulSet требует, чтобы вы создавали соответствующую управляющую headless-службу (то есть без кластерного IP-адреса службы), которая используется для предоставления фактической сетевой идентичности каждому модулю. Через эту службу каждый модуль получает свою собственную ресурсную запись DNS, чтобы соседи и, возможно, другие клиенты в кластере могли обращаться к модулю по его хостнейму имени. Например, если управляющая служба принадлежит пространству имен `default` и называется `foo`, а один из модулей называется `A-0`, то вы можете связаться с модулем через его FQDN-имя, то есть `a-0.foo.default.svc.cluster.local`. Вы не можете сделать это с модулями, управляемыми набором реплик.

Кроме того, вы также можете использовать DNS для поиска имен всех модулей набора StatefulSet путем поиска записей SRV для домена `foo.default.svc.cluster.local`. Мы дадим пояснения по поводу ресурсной записи SRV в разделе 10.4 и познакомимся с тем, как они используются для обнаружения членов набора StatefulSet.

Замена потерянных питомцев

Когда экземпляр модуля, управляемый набором StatefulSet, исчезает (поскольку узел, на котором этот модуль был запущен, был вытеснен из узла, либо кто-то удалил объект модуля вручную), набор StatefulSet обеспечивает его замену на новый экземпляр – подобно тому, как это делают наборы реплик ReplicaSet. Но, в отличие от наборов ReplicaSet, сменный модуль получает то же самое имя и хостнейм, что и модуль, который исчез (это различие между наборами ReplicaSet и наборами StatefulSet показано на рис. 10.6).

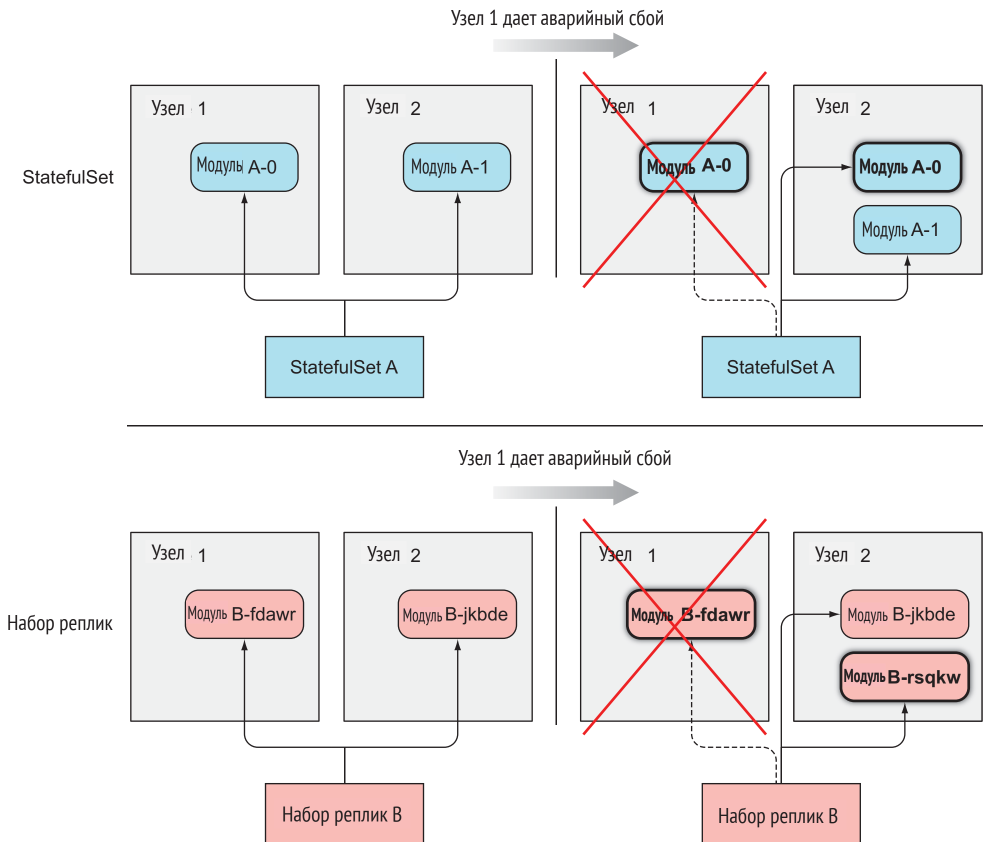


Рис. 10.6. Набор StatefulSet заменяет потерянный модуль на новый модуль с той же идентичностью, тогда как набор ReplicaSet заменяет его на совершенно новый несвязанный модуль

Новый модуль не обязательно назначен на тот же узел, но, как вы узнали на более раннем этапе, то, на каком узле работает модуль, не имеет значения. Это справедливо даже для модулей с внутренним состоянием. Даже если модуль назначен на другой узел, он все равно будет под тем же хостнеймом, что и раньше.

Масштабирование набора StatefulSet

При масштабировании набора модулей с внутренним состоянием StatefulSet создается новый экземпляр модуля с очередным неиспользованным порядковым индексом. Если вы увеличиваете масштаб с двух до трех экземпляров, то новый экземпляр получит индекс 2 (существующие экземпляры, разумеется, имеют индексы 0 и 1).

Приятная вещь относительно уменьшения масштаба набора StatefulSet состоит в том, что вы всегда знаете, какой модуль будет удален. Опять же, это также контрастирует с уменьшением масштаба наборов реплик ReplicaSet, где вы понятия не имеете, какой экземпляр будет удален, и вы даже не можете указать, какой из них вы хотите удалить первым (но этот функционал может быть внедрен в будущем). При уменьшении масштаба набора StatefulSet первыми удаляются экземпляры с наибольшим порядковым индексом (рис. 10.7). Это делает последствия уменьшения масштаба предсказуемыми.

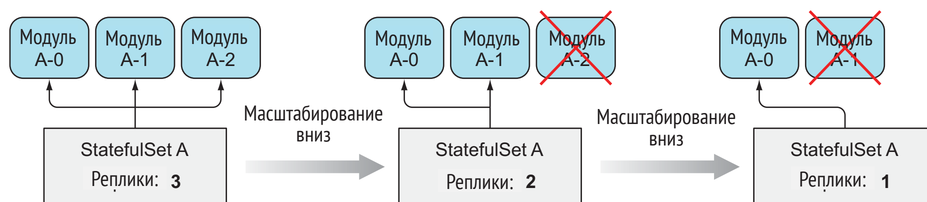


Рис. 10.7. При масштабировании вниз набора StatefulSet первым всегда удаляется модуль с наибольшим порядковым индексом

Поскольку некоторые приложения с внутренним состоянием не занимаются быстрым масштабированием вниз, наборы StatefulSet уменьшают масштаб только одного экземпляра модуля за один раз. Распределенное хранилище данных, к примеру, может потерять данные, если одновременно происходит аварийный сбой нескольких узлов. Например, если реплицируемое хранилище данных настроено для хранения двух копий каждой записи данных, в случаях, когда два узла одновременно выходят из строя, запись данных будет потеряна, если она хранится именно на этих двух узлах. Если уменьшение масштаба осуществляется последовательно, распределенное хранилище данных успевает создать дополнительную реплику записи данных в другом месте, чтобы заменить (единственную) потерянную копию.

По этой причине наборы StatefulSet также никогда не допускают операции масштабирования вниз, если какой-либо из экземпляров не здоров. Если экземпляр находится в неработоспособном состоянии и вы масштабируете вниз по одному одновременно, то вы практически потеряете сразу двух членов кластера.

10.2.3 Обеспечение стабильного выделенного хранилища для каждого экземпляра с внутренним состоянием

Вы видели, как наборы StatefulSet обеспечивают, чтобы модули с внутренним состоянием имели стабильную идентичность, а как насчет хранилища?

Каждый экземпляр модуля с внутренним состоянием должен использовать свое собственное хранилище, а также, если модуль с внутренним состоянием переназначен (заменен новым экземпляром, но с той же идентичностью, что и раньше), за новым экземпляром должно быть закреплено то же самое хранилище. Как добиться этого с помощью наборов модулей с внутренним состоянием StatefulSet?

Очевидно, хранилище для модулей с внутренним состоянием должно быть постоянным и отделенным от модулей. В главе 6 вы познакомились с постоянными томами PersistentVolume и заявками на получение постоянного тома PersistentVolumeClaim, которые позволяют закреплять постоянное хранилище за модулем, ссылаясь по имени на заявку PersistentVolumeClaim в модуле. Поскольку заявки PersistentVolumeClaim взаимно-однозначно соответствуют постоянным томам PersistentVolume, каждый модуль в наборе StatefulSet, для того чтобы иметь свой собственный отдельный постоянный том PersistentVolume, должен ссылаться на разные заявки PersistentVolumeClaim. Тогда каким образом каждый экземпляр модуля может ссылаться на разную заявку PersistentVolumeClaim, если все они штампуются из одного шаблона модуля? И кто создает эти заявки? Безусловно, от вас и не ожидается, что вы будете создавать столько заявок PersistentVolumeClaims, сколько будет модулей, которые вы планируете иметь в наборе StatefulSet. Разумеется, нет.

Кооперация шаблонов модулей с шаблонами заявок на тома

Набор StatefulSet должен создавать заявки PersistentVolumeClaim таким же образом, каким он создает модули. По этой причине набор StatefulSet может также иметь один или несколько шаблонов заявок на получение тома, которые позволяют ему штамповать заявки PersistentVolumeClaim (PVC) вместе с каждым экземпляром модуля (см. рис. 10.8).

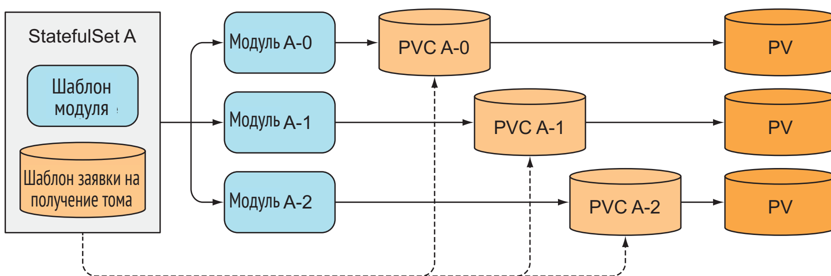


Рис. 10.8. Набор StatefulSet создает и модули, и заявки PersistentVolumeClaim

Постоянные тома PersistentVolume для заявок могут создаваться заранее администратором либо оперативно посредством динамического создания постоянных томов, как объяснено в конце главы 6.

Создание и удаление заявок на получение постоянных томов

Увеличение масштаба набора StatefulSet по одному модулю создает два или более объектов API (модуль и одну или несколько заявок PersistentVolumeClaim,

на которые ссылается модуль). При уменьшении масштаба, однако, удаляется только модуль, оставляя заявки в покое. Причина этого очевидна, если учесть, что именно происходит при удалении заявки. После удаления заявки постоянный том PersistentVolume, к которому она была привязана, рециркулируется либо удаляется, а его содержимое теряется.

Поскольку модули с внутренним состоянием предназначены для того, чтобы выполнять приложения с внутренним состоянием, из чего вытекает, что данные, которые они хранят в томе, представляют важность, удаление заявки при уменьшении масштаба набора StatefulSet может быть катастрофическим, в особенности если учесть, что инициирование уменьшения масштаба заключается в простом уменьшении поля `replicas` в описании ресурса StatefulSet. По этой причине, для того чтобы высвободить лежащий в основании постоянный том PersistentVolume, от вас требуется удалять заявки PersistentVolumeClaim вручную.

Повторное закрепление заявки PersistentVolumeClaim за новым экземпляром того же модуля

Тот факт, что заявка PersistentVolumeClaim остается после уменьшения масштаба, означает, что последующее увеличение масштаба может повторно закрепить ту же самую заявку вместе со связанным с ней постоянным томом PersistentVolume и его содержимым за новым экземпляром модуля (показано на рис. 10.9). Если вы уменьшили масштаб набора StatefulSet случайно, то вы можете отменить ошибку, снова увеличив масштаб, и новый модуль опять получит то же самое хранимое состояние (а также то же самое имя).

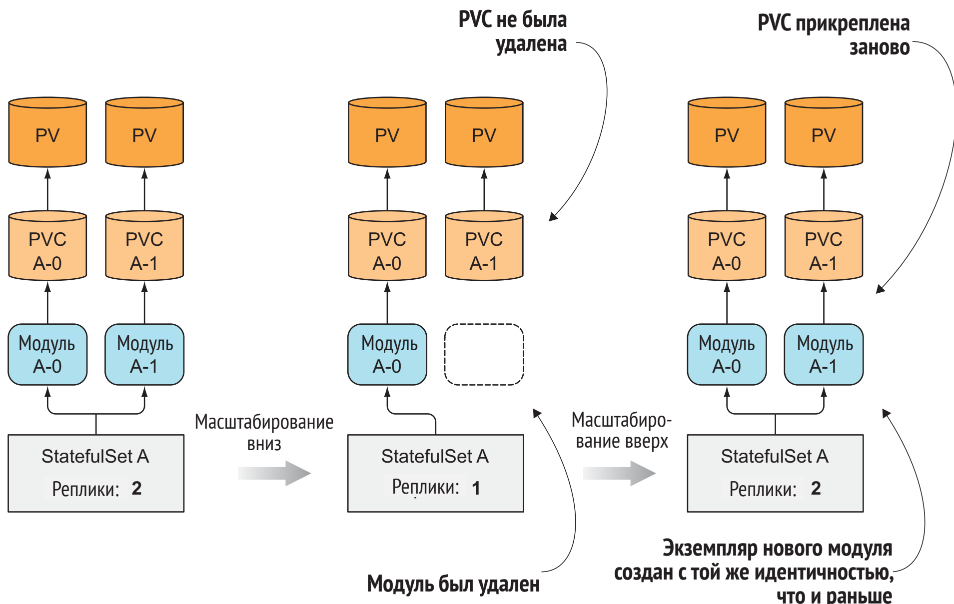


Рис. 10.9. При уменьшении масштаба наборы StatefulSet не удаляют заявки PersistentVolumeClaim (PVC); затем при увеличении масштаба они прикрепляют их заново

10.2.4 Гарантии набора StatefulSet

Как вы видели до сих пор, поведение наборов StatefulSet отличается от поведения наборов реплик ReplicaSet и контроллеров репликации ReplicationController. Но дело не заканчивается тем, что модули имеют стабильную идентичность и хранилище. Наборы StatefulSet также имеют отличающиеся гарантии относительно своих модулей.

Последствия стабильной идентичности и хранилища

В отличие от обычных модулей без внутреннего состояния, которые являются взаимозаменяемыми, модули с внутренним состоянием таковыми не являются. Мы уже видели, как модуль с внутренним состоянием всегда заменяется на идентичный модуль (с тем же именем и хостнеймом, с использованием того же постоянного хранилища и т. д.). Это происходит, когда Kubernetes видит, что старый модуль больше не существует (например, при удалении модуля вручную).

Но что, если Kubernetes не может быть уверен в состоянии модуля? Если он создает сменный модуль с одинаковой идентичностью, то в системе могут работать два экземпляра приложения с одинаковой идентичностью. Оба также будут привязаны к одному хранилищу, поэтому два процесса с одинаковой идентичностью будут перезаписывать одни и те же файлы. С модулями, управляемыми наборами реплик ReplicaSet, это не проблема, потому что приложения, безусловно, предназначены для работы с одними и теми же файлами. Кроме того, наборы ReplicaSet создают модули с произвольно сгенерированной идентичностью, поэтому ситуация, когда два процесса будут работать с одинаковой идентичностью, просто не может возникнуть.

Знакомство с семантикой «не более одного» набора StatefulSet

Следовательно, система Kubernetes должна тщательно позаботиться о том, чтобы два экземпляра модуля с внутренним состоянием гарантированно никогда не работали с одинаковой идентичностью и были связаны с одинаковым запросом на постоянный том PersistentVolumeClaim. Для экземпляров модуля с внутренним состоянием набор StatefulSet должен гарантировать семантику «не более одного».

Это означает, что, перед тем как набор StatefulSet создаст сменный модуль, он должен быть абсолютно уверен в том, что модуль больше не работает. А это оказывает большое влияние на то, как обрабатываются аварийные прекращения работы узлов. Мы продемонстрируем это позже в данной главе. Однако, прежде чем мы сможем это сделать, вам нужно создать набор StatefulSet и посмотреть, как он себя ведет. По ходу вы также узнаете еще несколько особенностей относительно их поведения.

10.3 Использование набора StatefulSet

Для того чтобы правильно показать наборы StatefulSet в действии, вы создадите собственное небольшое кластеризованное хранилище данных. Ничего необычного – оно больше похоже на древнее хранилище данных.

10.3.1 Создание приложения и образа контейнера

Вы будете применять приложение `kubia`, которое вы использовали в качестве отправной точки на протяжении всей книги. Вы расширите его, чтобы оно позволяло хранить и извлекать одну запись данных на каждом экземпляре модуля.

Важные фрагменты исходного кода хранилища данных приведены в следующем ниже листинге.

Листинг 10.1. Простое приложение с внутренним состоянием: kubia-pet-image/app.js

```
...
const dataFile = "/var/data/kubia.txt";
...
var handler = function(request, response) {
  if (request.method == 'POST') {
    var file = fs.createWriteStream(dataFile);
    file.on('open', function (fd) {
      request.pipe(file);
      console.log("New data has been received and stored.");
      response.writeHead(200);
      response.end("Data stored on pod " + os.hostname() + "\n");
    });
  } else {
    var data = fileExists(dataFile)
      ? fs.readFileSync(dataFile, 'utf8')
      : "No data posted yet";
    response.writeHead(200);
    response.write("You've hit " + os.hostname() + "\n");
    response.end("Data stored on this pod: " + data + "\n");
  }
};

var www = http.createServer(handler);
www.listen(8080);
```

При отправке запросов POST сохранить тело запроса в файл данных

В запросах GET (и всех других типах запросов) возвращать ваш хостнейм и содержимое файла данных

Всякий раз, когда приложение получает запрос POST, оно записывает данные, полученные в теле запроса, в файл /var/data/kubia.txt. По запросу GET оно возвращает хостнейм и сохраненные данные (содержимое файла). Достаточно просто, верно? Это первая версия приложения. Он еще не кластеризовано, но этого достаточно, чтобы начать. Вы развернете приложение позже в этой главе.

Файл Dockerfile для построения образа контейнера показан в следующем ниже листинге и остался прежним.

Листинг 10.2. Файл Dockerfile для приложения с внутренним состоянием: kubia-pet-image/Dockerfile

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

Продолжайте и теперь соберите образ либо используйте тот, который я отправил в docker.io/luksa/kubia-pet.

10.3.2 Развертывание приложения посредством набора StatefulSet

Для того чтобы развернуть приложение, необходимо создать два (или три) разных типа объектов:

- объекты PersistentVolume для хранения файлов данных (их необходимо создавать только в том случае, если кластер не поддерживает динамическое резервирование постоянных томов PersistentVolume);
- управляющая служба, требуемая набором StatefulSet;
- сам набор с внутренним состоянием StatefulSet.

Для каждого экземпляра модуля объект StatefulSet создаст запрос PersistentVolumeClaim, который привяжет к постоянному тому PersistentVolume. Если кластер поддерживает динамическое создание, то создавать постоянные тома PersistentVolume вручную не требуется (и следующий ниже раздел можно пропустить). Если это не так, то вам нужно их создать, как описано в следующем далее разделе.

Создание постоянных томов

Вам понадобится три постоянных тома PersistentVolume, потому что вы будете масштабировать StatefulSet вверх до трех реплик. Вы должны создать больше, если планируете масштабирование StatefulSet выше.

Если вы используете Minikube, разверните постоянные тома, определенные в файле Chapter06/persistent-volumes-hostpath.yaml в архиве кода, прилагаемого к этой книге.

Если вы используете движок Google Kubernetes Engine, то вам сначала нужно фактически создать постоянные диски GCE GCE Persistent Disk, как показано ниже:

```
$ gcloud compute disks create --size=1GiB --zone=europe-west1-b pv-a
$ gcloud compute disks create --size=1GiB --zone=europe-west1-b pv-b
$ gcloud compute disks create --size=1GiB --zone=europe-west1-b pv-c
```

ПРИМЕЧАНИЕ. Проверьте, чтобы диски создавались в той же зоне, в которой работают ваши узлы.

Затем создайте постоянные тома из файла persistent-volumes-gcepd.yaml, который показан в следующем ниже листинге.

Листинг 10.3. Три постоянных тома: persistent-volumes-gcepd.yaml

```
kind: List                                  <-  Файл описывает список
apiVersion: v1                                  из трех постоянных томов
items:
- apiVersion: v1
  kind: PersistentVolume                    <-
  metadata:                                     Имена постоянных
    name: pv-a                              <-  томов-pv-a, pv-b и pv-c
  spec:
    capacity:                                   Емкость каждого постоянного
      storage: 1Mi                          <-  тома – 1 мебибайт
    accessModes:
      - ReadWriteOnce                           Когда том освобождается заявкой,
    persistentVolumeReclaimPolicy: Recycle  <-  он рециркулируется для повторного
    gcePersistentDisk:                          использования
      pdName: pv-a
      fsType: nfs4                          <-  Том использует постоянный диск GCE
- apiVersion: v1                                Persistent Disk в качестве базового
  kind: PersistentVolume                        механизма хранения
  metadata:
    name: pv-b
...
```

> **ПРИМЕЧАНИЕ.** В предыдущей главе вы указали несколько ресурсов в одном YAML, разделяя их строкой из трех тире. Здесь вы используете другой подход, определяя объект-список `List` и перечисляя ресурсы как элементы объекта. Оба метода эквивалентны.

Этот манифест создает постоянные тома PersistentVolumes под названием `pv-a`, `pv-b` и `pv-c`. В качестве базового механизма хранения они используют постоянные диски GCE Persistent Disk, поэтому они не подходят для кластеров, которые не работают на Google Kubernetes Engine или Google Compute Engine. При запуске кластера в другом месте необходимо изменить определение ресурса PersistentVolume и использовать соответствующий тип тома, например NFS (сетевая файловая система) или аналогичный.

Создание управляющей службы

Как отмечалось ранее, перед развертыванием набора StatefulSet сначала необходимо создать службу без обозначенной точки входа (Headless), которая будет использоваться для предоставления сетевой идентичности для модулей с внутренним состоянием. В следующем ниже листинге показан манифест этой службы.

Листинг 10.4. Служба без обозначенной точки входа, которая будет использоваться в наборе StatefulSet: kubia-service-headless.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia          <- Имя службы
spec:
  clusterIP: None      <- Управляющая служба набора StatefulSet должна быть без управляемой точки входа
  selector:
    app: kubia         <- Этой службе принадлежат все модули с меткой app=kubia
  ports:
  - name: http
    port: 80
```

Вы выставляете поле `clusterIP` в `None`, что делает эту службу службой без обозначенной точки входа. Это позволит вашим модулям обнаруживать соседей в наборе. После создания службы можно перейти к созданию фактического набора модулей с внутренним состоянием StatefulSet.

Создание манифеста набора StatefulSet

Теперь вы можете, наконец, создать ресурс StatefulSet. В следующем ниже листинге показан его манифест.

Листинг 10.5. Ресурс StatefulSet: kubia-statefulset.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: kubia
spec:
  serviceName: kubia
  replicas: 2
  template:
    metadata:
      labels:
        app: kubia     <- Модули, созданные объектом StatefulSet, будут иметь метку app=kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia-pet
        ports:
        - name: http
          containerPort: 8080

        volumeMounts:
```

```
        - name: data
          mountPath: /var/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      resources:
        requests:
          storage: 1Mi
      accessModes:
      - ReadWriteOnce
```

Контейнер внутри модуля установит том pvc в этом пути

Заявки PersistentVolumeClaim будут созданы из этого шаблона

Манифест набора StatefulSet не отличается от создаваемых до сих пор манифестов наборов ReplicaSet или развертываний Deployment. Новое здесь – это список шаблонов `volumeClaimTemplates`. В нем вы определяете один шаблон заявки на постоянный том под названием `data`, который будет использоваться для создания заявки PersistentVolumeClaim для каждого модуля. Как вы помните из главы 6, модуль ссылается на заявку, включая в манифест том `persistentVolumeClaim`. В предыдущем шаблоне модуля вы не найдете такого тома. Набор StatefulSet автоматически добавляет его в спецификацию модуля и настраивает том, который будет привязан к заявке, которую набор StatefulSet создал для конкретного модуля.

Создание набора модулей с внутренним состоянием StatefulSet

Теперь создайте набор StatefulSet:

```
$ kubectl create -f kubia-statefulset.yaml
statefulset "kubia" created
```

И выведите список своих модулей:

```
$ kubectl get po
NAME     READY  STATUS             RESTARTS  AGE
kubia-0  0/1    ContainerCreating  0         1s
```

Ничего не замечаете? Помните, как объекты ReplicationController или ReplicaSet создают все экземпляры модуля одновременно? Ваш объект StatefulSet настроен на создание двух реплик, но он создал один-единственный модуль.

Не переживайте, все в порядке. Второй модуль будет создан только после того, как первый будет готов. Объекты StatefulSet ведут себя таким образом, потому что некоторые кластеризованные приложения с внутренним состоянием чувствительны к «race condition», если два или более членов кластера появляются в то же самое время, поэтому безопаснее поднимать каждого члена полностью и только потом продолжать поднятие остальных.

Еще раз выведем список, для того чтобы увидеть, как развивается создание модулей:

```
$ kubectl get po
NAME     READY  STATUS             RESTARTS  AGE
kubia-0  1/1    Running            0         8s
kubia-1  0/1    ContainerCreating  0         2s
```

Видите, первый модуль запущен, а второй создан и только запускается.

Исследование сгенерированного модуля с внутренним состоянием

Давайте более подробно рассмотрим спецификацию первого модуля в следующем ниже листинге, чтобы увидеть, как объект StatefulSet сконструировал модуль из шаблона модуля и шаблона заявки PersistentVolumeClaim.

Листинг 10.6. Модуль с внутренним состоянием, созданный объектом StatefulSet

```
$ kubectl get po kubia-0 -o yaml
apiVersion: v1
kind: Pod
metadata:
  ...
spec:
  containers:
  - image: luksa/kubia-pet
    ...
    volumeMounts:
    - mountPath: /var/data          <- Монтирование тома, как указано в манифесте
      name: data
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-r2m41
      readOnly: true
  ...
  volumes:
  - name: data                      <- Том, созданный объектом StatefulSet
    persistentVolumeClaim:
      claimName: data-kubia-0       <- Заявка, на которую ссылается этот том
  - name: default-token-r2m41
    secret:
      secretName: default-token-r2m41
```

Шаблон заявки PersistentVolumeClaim использовался для создания заявки PersistentVolumeClaim и тома внутри модуля, который ссылается на созданную заявку PersistentVolumeClaim.

Исследование сгенерированных заявок PersistentVolumeClaim

Теперь выведем список сгенерированных заявок PersistentVolumeClaim, чтобы подтвердить, что они были созданы:

```
$ kubectl get pvc
NAME          STATUS  VOLUME  CAPACITY  ACCESSMODES  AGE
data-kubia-0  Bound   pv-c    0                      37s
data-kubia-1  Bound   pv-a    0                      37s
```

Имена сгенерированных заявок PersistentVolumeClaim скомпонованы из имени, определенного в шаблоне `volumeClaimTemplate`, и имени каждого модуля. Вы можете исследовать YAML заявок, чтобы убедиться, что они соответствуют шаблону.

10.3.3 Исследование своих модулей

Теперь, когда узлы кластера хранилища данных запущены, можно приступать к его изучению. Вы не можете обмениваться с вашими модулями через службу, которую вы создали, потому что она не имеет явной точки входа. Вам нужно будет напрямую подключаться к отдельным модулям (или создать обычную службу, но это не позволит вам обмениваться с конкретным модулем).

Вы уже видели способы подключения к модулю напрямую: заходя на соседний модуль и запуская внутри него утилиту `curl`, используя переадресацию портов и т. д. На этот раз вы попробуете еще один вариант. В качестве прокси для модулей вы будете использовать сервер API.

Взаимодействие с модулями через сервер API

Одной из полезных функциональных особенностей сервера API является возможность прокси-соединений непосредственно с отдельными модулями. Если вы хотите делать запросы к модулю `kubia-0`, перейдите по следующему URL-адресу:

```
<хостСервераApi>:<порт>/api/v1/namespaces/default/pods/kubia-0/proxy/<путь>
```

Поскольку сервер API защищен, отправка запросов в модули через сервер API будет громоздкой (помимо прочего, в каждом запросе необходимо передавать токен авторизации). К счастью, в главе 8 вы узнали, как для общения с сервером API использовать команду `kubectl proxy` без необходимости иметь дело с аутентификацией и сертификатами SSL. Запустите прокси снова:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Теперь, поскольку вы будете обмениваться с сервером API через `kubectl proxy`, вместо фактического хоста и порта сервера API вы будете использовать localhost:8001. Вы отправляете запрос в модуль `kubia-0` следующим образом:

```
$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
You've hit kubia-0
Data stored on this pod: No data posted yet
```

Отклик показывает, что запрос действительно был получен и обработан приложением, запущенным в вашем модуле `kubia-0`.

> **ПРИМЕЧАНИЕ.** Если вы получаете пустой отклик, убедитесь, что вы не упустили последний слеш в конце URL-адреса (или убедитесь, что утилита `curl` следует переадресациям с использованием своего параметра `-L`).

Поскольку вы обмениваетесь с модулем через сервер API, к которому вы подключаетесь через прокси `kubectl`, запрос прошел через два разных прокси (первым был прокси `kubectl`, и другим был сервер API, который проксировал запрос в модуль). Для получения более четкой картины изучите рис. 10.10.

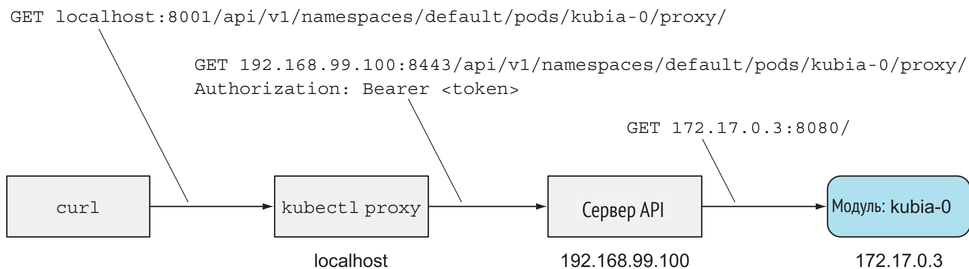


Рис. 10.10. Подключение к модулю через прокси kubectl и прокси сервера API

Запрос, отправленный в модуль, был запросом GET, но вы также можете через сервер API отправлять запросы POST. Это делается путем отправки запроса POST на тот же URL-адрес прокси, на который был отправлен запрос GET.

Когда приложение получает запрос POST, оно сохраняет все содержимое тела запроса в локальный файл. Отправьте запрос POST в модуль `kubia-0`:

```
$ curl -X POST -d "Hey there! This greeting was submitted to kubia-0."
➥  localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
Data stored on pod kubia-0
```

Отправленные данные теперь должны храниться в этом модуле. Давайте посмотрим, возвращает ли он сохраненные данные при повторном выполнении запроса GET:

```
$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
You've hit kubia-0
Data stored on this pod: Hey there! This greeting was submitted to kubia-0.
```

Ладно, пока все хорошо. Теперь давайте посмотрим, что говорит другой узел кластера (модуль `kubia-1`):

```
$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-1/proxy/
You've hit kubia-1
Data stored on this pod: No data posted yet
```

Как и ожидалось, каждый узел имеет свое собственное состояние. Но сохраняется ли это состояние? Давайте это выясним.

Удаление модуля с внутренним состоянием, чтобы убедиться, что переназначенный модуль прикреплен к тому же хранилищу

Вы собираетесь удалить модуль `kubia-0` и подождать, пока он не будет переназначен. Затем вы увидите, раздает ли он по-прежнему те же самые данные, как и раньше:

```
$ kubectl delete po kubia-0
pod "kubia-0" deleted
```

Если вывести список модулей, то вы увидите, что модуль прекращает свою работу:

```
$ kubectl get po
NAME     READY  STATUS       RESTARTS  AGE
kubia-0  1/1    Terminating  0         3m
kubia-1  1/1    Running      0         3m
```

Как только он успешно прекратит работу, объект StatefulSet создаст новый модуль с тем же именем:

```
$ kubectl get po
NAME        READY  STATUS             RESTARTS  AGE
kubia-0     0/1    ContainerCreating  0         6s
kubia-1     1/1    Running 0 4m
$ kubectl get po
NAME        READY  STATUS   RESTARTS  AGE
kubia-0     1/1    Running  0         9s
kubia-1     1/1    Running  0         4m
```

Следует напомнить еще раз, что этот новый модуль может быть назначен на любой узел в кластере, а не обязательно на тот же узел, что и у старого модуля. Вся идентичность старого модуля (имя, хостнейм и хранилище) фактически перемещена в новый узел (как показано на рис. 10.11). Если вы используете Minikube, то вы это не увидите, потому что он управляет только одним узлом, но в кластере с несколькими узлами вы можете увидеть модуль, назначенный на узел, который отличается от того, что был раньше.

Теперь, когда новый модуль запущен, давайте проверим, имеет ли он ту же идентичность, что и в предыдущем воплощении. Имя модуля совпадает, но как насчет сетевого имени и устойчивых данных? Для того чтобы получить подтверждение, вы можете запросить непосредственно сам модуль:

```
$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
You've hit kubia-0
Data stored on this pod: Hey there! This greeting was submitted to kubia-0.
```

Отклик модуля показывает, что и сетевое имя, и данные совпадают, как и раньше, подтверждая, что объект StatefulSet всегда заменяет удаленный модуль на тот, который на самом деле является тем же самым модулем.

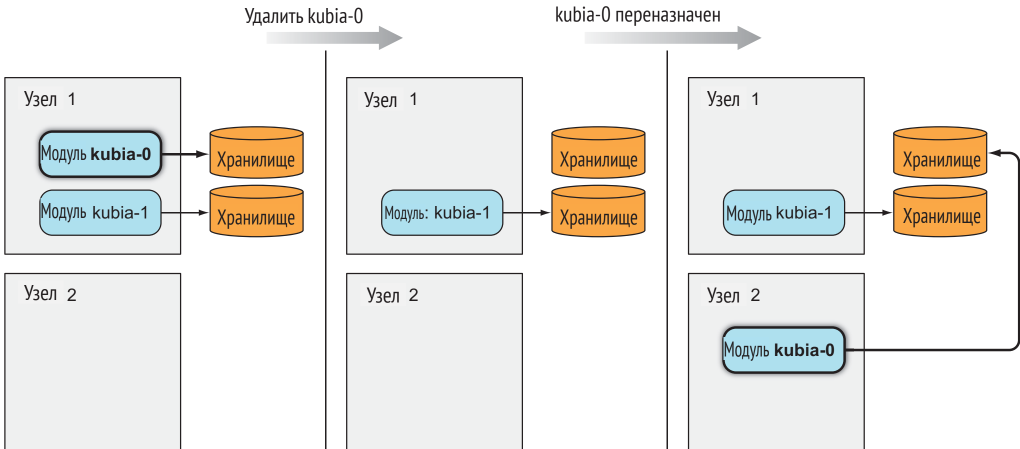


Рис. 10.11. Модуль с внутренним состоянием может быть переназначен на другой узел, но он сохраняет имя, сетевое имя и хранилище

Масштабирование объекта StatefulSet

Масштабирование объекта StatefulSet вниз и его масштабирование назад вверх после длительного периода времени не должно отличаться от удаления модуля и затем немедленного его воссоздания объектом StatefulSet. Помните, что, масштабируя вниз, объект StatefulSet только удаляет модули, но оставляет заявки PersistentVolumeClaim нетронутыми. Вы можете сами попробовать уменьшить масштаб объект StatefulSet и получить подтверждение этому поведению.

Главное – запомнить, что масштабирование вниз (и вверх) выполняется постепенно – подобно тому, как отдельные модули создаются при первоначальном создании объекта StatefulSet. При уменьшении масштаба более чем на один экземпляр модуль с наибольшим порядковым номером удаляется первым. Модуль со вторым самым высоким порядковым номером удаляется только после того, как предыдущий модуль полностью завершит свою работу.

Предоставление доступа к модулям с внутренним состоянием через службу без заданной точки входа

Перед тем как перейти к последней части этой главы, вы добавите правильную, Headless-службу, которая будет предварять ваши модули, потому что клиенты обычно подключаются к модулям через службу, а не напрямую.

К настоящему времени вы уже знаете, как создавать службу, но в случае если это не так, то следующий ниже листинг показывает соответствующий манифест.

Листинг 10.7. Обычная служба для доступа к модулям с внутренним состояниям: kubia-service-public.yaml

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: kubia-public
spec:
  selector:
    app: kubia
  ports:
  - port: 80
    targetPort: 8080
```

Поскольку это не внешняя служба (это обычная служба `ClusterIP`, а не служба типа `NodePort` или `LoadBalancer`), доступ к ней можно получить только из кластера. Для того чтобы получить к ней доступ, понадобится модуль, верно? Необязательно.

Подключение к внутренним службам кластера через сервер API

Вместо того чтобы использовать проходной модуль для доступа к службе из кластера, вы можете воспользоваться тем же функционалом прокси, обеспечиваемым сервером API для доступа к службе таким образом, каким вы обращались к отдельным модулям.

Путь в URI-адресе для проксирования запросов к службам формируется следующим образом:

```
/api/v1/namespaces/<пространство имен>/services/<имя службы>/proxy/<путь>
```

Поэтому вы можете применить утилиту `curl` на локальном компьютере и получить доступ к службе через прокси `kubectl` следующим образом (вы применили `kubectl proxy` ранее, и он должен по-прежнему работать):

```
$ curl localhost:8001/api/v1/namespaces/default/services/kubia-
➥  public/proxy/
You've hit kubia-1
Data stored on this pod: No data posted yet
```

Аналогично этому, клиенты (внутри кластера) могут использовать службу `kubia-public` для сохранения и чтения данных из кластеризованного хранилища данных. Разумеется, каждый запрос попадает на случайный узел кластера, поэтому каждый раз вы будете получать данные из случайного узла. Вы это усовершенствуете в следующий раз.

10.4 Обнаружение соседей в наборе StatefulSet

Нам осталось осветить еще один важный аспект. Важным требованием кластеризованных приложений является обнаружение соседей – умение находить других членов кластера. Каждый член набора StatefulSet должен легко находить всех других членов. Конечно, это можно сделать, обменявшись с сервером API, но одна из целей Kubernetes – предоставить функционал, который

помогает поддерживать приложения полностью платформенно-независимыми от Kubernetes. Следовательно, нежелательно, чтобы приложения обменивались с API Kubernetes.

Каким образом модуль может обнаружить свои одноранговые модули, не обмениваясь с API? Есть ли существующая, известная технология, делающая это возможным, которую можно было бы задействовать? Как насчет системы доменных имен (DNS)? В зависимости от того, как много вы знаете о DNS, вы, вероятно, понимаете, для чего используется ресурсная запись A, CNAME или MX. Существуют и другие менее известные типы ресурсных записей DNS. Одна из них – запись SRV.

Знакомство с записью SRV

Записи SRV используются для указания на хостнеймы и порты серверов, предоставляющих конкретную службу. Kubernetes создает записи SRV, чтобы указывать на хостнеймы модулей, поддерживающие службы, без явного указания точки входа.

Вы можете получить список записей SRV для ваших модулей с внутренним состоянием, запустив инструмент поиска в DNS `dig` внутри нового временного модуля. Вот команда, которую вы будете использовать:

```
$ kubectl run -it srvlookup --image=tutum/dnsutils --rm
➥  --restart=Never -- dig SRV kubia.default.svc.cluster.local
```

Команда запускает одноразовый модуль (`--restart=Never`) с именем `srvlookup`, который прикрепляется к консоли (`-it`) и удаляется, как только он завершается (`--rm`). Этот модуль запускает единственный контейнер из образа `tutum/dnsutils` и выполняет следующую ниже команду:

```
dig SRV kubia.default.svc.cluster.local
```

В следующем ниже листинге показано, что именно эта команда выводит на печать.

Листинг 10.8. Листинг ресурсных записей SRV системы доменных имен (DNS) вашей безголовой службы

```
...
;; ANSWER SECTION:
k.d.s.c.l. 30 IN SRV 10 33 0 kubia-0.kubia.default.svc.cluster.local.
k.d.s.c.l. 30 IN SRV 10 33 0 kubia-1.kubia.default.svc.cluster.local.

;; ADDITIONAL SECTION:
kubia-0.kubia.default.svc.cluster.local. 30 IN A 172.17.0.4
kubia-1.kubia.default.svc.cluster.local. 30 IN A 172.17.0.6
...
```

> **ПРИМЕЧАНИЕ.** Мне пришлось сократить фактическое имя, с тем чтобы записи поместились в одной строке. Так, `kubia.d.s.c.l` на самом деле имеет вид `kubia.default.svc.cluster.local`.

В разделе ответов `ANSWER SECTION` показаны две ресурсные записи `SRV`, указывающие на два модуля, поддерживающих службу, без явного указания точки входа. Каждый модуль также получает свою собственную запись. Это показано в дополнительном разделе `ADDITIONAL SECTION`.

Для того чтобы модуль получил список всех других модулей набора StatefulSet, вам всего лишь нужно выполнить поиск SRV в DNS. В Node.js, например, данный поиск выполняется следующим образом:

```
dns.resolveSrv("kubia.default.svc.cluster.local", callBackFunction);
```

Вы будете применять эту команду в своем приложении, чтобы позволить каждому модулю обнаруживать своих соседей.

> **ПРИМЕЧАНИЕ.** Порядок возвращаемых ресурсных записей SRV является случайным, поскольку все они имеют одинаковый приоритет. Не ожидайте, что в списке всегда будете видеть `kubia-0` перед `kubia-1`.

10.4.1 Реализация обнаружения соседей посредством DNS

Ваше древнее хранилище данных еще не кластеризовано. Каждый узел хранилища данных работает полностью независимо от всех остальных – никакой связи между ними не существует. Вы заставите их обмениваться друг с другом.

Данные, отправляемые клиентами, подключающимися к кластеру хранилища данных через службу `kubia-public`, размещаются на случайном узле кластера. Кластер может хранить несколько записей данных, но клиенты в настоящее время не имеют хорошего способа видеть все эти записи. Если клиент захочет получить данные из всех модулей, то поскольку службы перенаправляют запросы в модули случайным образом, то прежде чем он достигнет всех модулей, ему потребуется выполнить много запросов.

Это положение можно улучшить, если узел будет откликаться данными со всех узлов кластера. Для этого узел должен найти всех своих соседей. И для этого вы будете использовать то, что вы узнали о наборах StatefulSet и ресурсных записях SRV.

Вы измените исходный код приложения, как показано в следующем ниже листинге (полный исходный код доступен в архиве кода, прилагаемого к этой книге; в листинге показаны только важные фрагменты).

Листинг 10.9. Обнаружение одноранговых модулей в примере приложения: kubia-pet-peers-image/app.js

```
...
const dns = require('dns');

const dataFile = "/var/data/kubia.txt";
const serviceName = "kubia.default.svc.cluster.local";
const port = 8080;
...
var handler = function(request, response) {
  if (request.method == 'POST') {
    ...
  } else {
    response.writeHead(200);
    if (request.url == '/data') {
      var data = fileExists(dataFile)
        ? fs.readFileSync(dataFile, 'utf8')
        : "No data posted yet";
      response.end(data);
    } else {
      response.write("You've hit " + os.hostname() + "\n");
      response.write("Data stored in the cluster:\n");
      dns.resolveSrv(serviceName, function (err, addresses) {
        if (err) {
          response.end("Could not look up DNS SRV records: " + err);
          return;
        }
        var numResponses = 0;
        if (addresses.length == 0) {
          response.end("No peers discovered.");
        } else {
          addresses.forEach(function (item) {
            var requestOptions = {
              host: item.name,
              port: port,
              path: '/data'
            };
            httpGet(requestOptions, function (returnedData) {
              numResponses++;
              response.write("- " + item.name + ": " + returnedData);
              response.write("\n");
              if (numResponses == addresses.length) {
                response.end();
              }
            });
```

Приложение выполняет поиск в DNS для получения ресурсных записей SRV

Каждый модуль, на который указала запись SRV, после этого адресуется, для того чтобы получить его данные

```
          });
        }
      });
    }
  }
};
...
```

На рис. 10.12 показано, что происходит, когда приложение получает запрос GET. Сервер, который получает запрос, сначала выполняет поиск ресурсных записей SRV для службы, без явного указания точки входа `kubia`, а затем отправляет запрос GET в каждый модуль, поддерживающий службу (даже самому себе, что, очевидно, не нужно, но я хотел, чтобы код был максимально простым). Затем он возвращает список всех узлов вместе с данными, хранящимися на каждом из них.

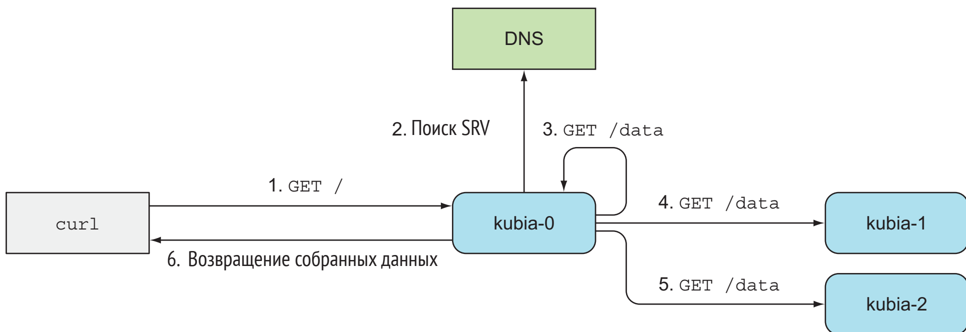


Рис. 10.12. Работа упрощенного распределенного хранилища данных

Образ контейнера, содержащий эту новую версию приложения, доступен в docker.io/luksa/kubia-pet-peers.

10.4.2 Обновление набора StatefulSet

Ваш набор StatefulSet уже запущен, поэтому давайте посмотрим, как обновить его шаблон модуля, чтобы модули использовали этот новый образ. Одновременно с этим вы также выставите количество реплик в 3. Для того чтобы обновить набор StatefulSet, используйте команду `kubectl edit` (команда `patch` будет еще одним вариантом):

```
$ kubectl edit statefulset kubia
```

Определение ресурса StatefulSet откроется в редакторе, выбранном по умолчанию. В определении поменяйте `spec.replicas` на 3 и измените атрибут `spec.template.spec.containers.image` так, чтобы он указывал на новый образ (`luksa/kubia-pet-peers` вместо `luksa/kubia-pet`). Сохраните файл и закройте редактор, чтобы обновить набор StatefulSet. Две реплики были запущены ранее, поэтому теперь вы должны увидеть, как запускается дополнительная

реплика под названием kubia-2. Выведите список модулей для получения подтверждения:

```
$ kubectl get po
NAME     READY  STATUS             RESTARTS  AGE
kubia-0  1/1    Running            0         25m
kubia-1  1/1    Running            0         26m
kubia-2  0/1    ContainerCreating  0         4s
```

Новый экземпляр модуля выполняет новый образ. Но как насчет существующих двух реплик? Судя по их возрасту, они, кажется, не были обновлены. Это ожидаемо, поскольку изначально наборы StatefulSet были больше похожи на наборы реплик, а не на развертывания, поэтому при изменении шаблона они развертывания не выполняют. Необходимо удалить реплики вручную, и набор StatefulSet снова вызовет их на основе нового шаблона:

```
$ kubectl delete po kubia-0 kubia-1
pod "kubia-0" deleted
pod "kubia-1" deleted
```

> **ПРИМЕЧАНИЕ.** Начиная с Kubernetes версии 1.7 наборы StatefulSet поддерживают плавные обновления точно так же, как это делают развертывания Deployment и наборы DaemonSet. Относительно более подробной информации обратитесь к документации по полю `spec.updateStrategy` набора StatefulSet, используя для этого команду `kubectl explain`.

10.4.3 Опробование кластеризованного хранилища данных

После того как два модуля подняты, вы можете увидеть, работает ли ваше блестящее новое древнее хранилище данных должным образом. Отправьте несколько запросов в кластер, как показано в следующем ниже листинге.

Листинг 10.10. Запись в кластеризованное хранилище данных через службу

```
$ curl -X POST -d "The sun is shining" \
➥  localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/
Data stored on pod kubia-1

$ curl -X POST -d "The weather is sweet" \
➥  localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/
Data stored on pod kubia-0
```

Теперь прочитайте сохраненные данные, как показано в следующем ниже листинге.

Листинг 10.11. Чтение из хранилища данных

```
$ curl localhost:8001/api/v1/namespaces/default/services
➥  /kubia-public/proxy/
You've hit kubia-2
Data stored on each cluster node:
- kubia-0.kubia.default.svc.cluster.local: The weather is sweet
- kubia-1.kubia.default.svc.cluster.local: The sun is shining
- kubia-2.kubia.default.svc.cluster.local: No data posted yet
```

Замечательно! Когда клиентский запрос достигает одного из узлов кластера, тот обнаруживает всех его соседей, собирает из них данные и отправляет все данные обратно клиенту. Даже если вы масштабируете набор StatefulSet вверх или вниз, модуль, обслуживающий запрос клиента, всегда может найти все узлы, работающие в то время.

Само по себе приложение не очень полезно, но я надеюсь, что вы нашли интересный способ показать, как экземпляры реплицированного приложения с внутренним состоянием могут обнаруживать свои одноранговые модули и легко справляться с горизонтальным масштабированием.

10.5 Как наборы StatefulSet справляются с аварийными сбоями узлов

В разделе 10.2.4 мы отметили, что Kubernetes, прежде чем создавать замену модулю, должен быть абсолютно уверен, что модуль с внутренним состоянием больше не работает. Когда узел внезапно выходит из строя, Kubernetes не может знать состояние узла или его модулей. Он не может знать, работают ли модули вообще, или же они по-прежнему работают и, возможно, даже по-прежнему достижимы, и только Kubelet перестал сообщать о состоянии узла ведущему узлу.

Поскольку набор StatefulSet гарантирует, что никогда не будет двух модулей, работающих с одной и той же идентичностью и хранилищем, при аварийном сбое узла набор StatefulSet не может и не должен создавать сменный модуль до тех пор, пока не будет точно известно, что модуль больше не работает.

Он может узнать это только тогда, когда об этом сообщит администратор кластера. Для этого админ должен удалить модуль или удалить весь узел (делая так, затем удаляются все модули, приписанные к узлу).

В последнем упражнении этой главы вы узнаете, что происходит с наборами StatefulSet и их модулями, когда один из узлов кластера отключается от сети.

10.5.1 Симулирование отключения узла от сети

Как и в главе 4, вы будете симулировать отключения узла от сети, выключая сетевой интерфейс `eth0` узла. Поскольку в этом примере требуется несколько

узлов, вы не сможете выполнить его на Minikube. Вместо этого вы будете использовать Google Kubernetes Engine.

Завершение работы сетевого адаптера узла

Для того чтобы завершить работу интерфейса `eth0` узла, вам нужно войти по `ssh` в один из узлов, как показано ниже:

```
$ gcloud compute ssh gke-kubia-default-pool-32a2cac8-m0g1
```

Затем в узле выполните следующую ниже команду:

```
$ sudo ifconfig eth0 down
```

Ваш сеанс `ssh` перестанет работать, поэтому, для того чтобы продолжить, вам нужно будет открыть еще один терминал.

Проверка состояния узла с точки зрения ведущего узла Kubernetes

При отключенном сетевом интерфейсе узла работающий на узле Kubelet больше не сможет связываться с сервером API Kubernetes и сообщать ему, что узел и все его модули по-прежнему работают.

Через некоторое время плоскость управления пометит узел как неготовый, `NotReady`. Это можно увидеть при выведении списка узлов, как показано в следующем ниже листинге.

Листинг 10.12. Изменение статуса аварийно отказавшего узла на NotReady

```
$ kubectl get node
NAME                                  STATUS    AGE  VERSION
gke-kubia-default-pool-32a2cac8-596v  Ready     16m  v1.6.2
gke-kubia-default-pool-32a2cac8-m0g1  NotReady  16m  v1.6.2
gke-kubia-default-pool-32a2cac8-sgl7  Ready     16m  v1.6.2
```

Поскольку плоскость управления больше не получает обновления статуса от узла, статус всех модулей на этом узле неизвестен. Это показано в списке модулей в следующем ниже листинге.

Листинг 10.13. Изменение статуса модуля, после того как его узел становится неготовым

```
$ kubectl get po
NAME         READY    STATUS   RESTARTS  AGE
kubia-0 1/1  Unknown           0         15m
kubia-1 1/1  Running           0         14m
kubia-2 1/1  Running           0         13m
```

Как вы можете видеть, статус модуля `kubia-0` больше неизвестен, потому что этот модуль был (и по-прежнему) запущен на узле, сетевой интерфейс которого вы отключили.

Что происходит с модулями, статус которых неизвестен

Если бы узел вернулся в работу и сообщил о своем статусе и статусах своих модулей, то модуль снова был бы отмечен как работающий, `Running`. Но если статус модуля остается неизвестным более нескольких минут (это время настраивается), то модуль автоматически вытесняется из узла. Это делается ведущим узлом, мастером (плоскостью управления Kubernetes). Он вытесняет модуль, удаляя ресурс модуля.

Когда Kubelet видит, что модуль был помечен на удаление, он начинает завершать работу модуля. В вашем случае Kubelet больше не может достигнуть ведущего узла (потому что вы отключили узел от сети), а это значит, что модуль будет продолжать работать.

Рассмотрим текущую ситуацию. Используйте команду `kubectl describe` для вывода на экран сведений о модуле `kubia-0`, как показано в следующем ниже листинге.

Листинг 10.14. Вывод сведений о модуле с неизвестным статусом

```
$ kubectl describe po kubia-0
Name:         kubia-0
Namespace:    default
Node:         gke-kubia-default-pool-32a2cac8-m0g1/10.132.0.2
...
Status:       Terminating (expires Tue, 23 May 2017 15:06:09 +0200)
Reason:       NodeLost
Message:      Node gke-kubia-default-pool-32a2cac8-m0g1 which was
              running pod kubia-0 is unresponsive
```

Модуль показан как завершающий работу, `Terminating`, с причиной для прекращения – `NodeLost`, узел потерян. Это сообщение говорит о том, что узел считается потерянным, потому что он не откликается.

> **ПРИМЕЧАНИЕ.** Здесь показано представление о мире с точки зрения плоскости управления. На самом деле контейнер модуля по-прежнему отлично работает. И он вовсе не заканчивает работу.

10.5.2 Удаление модуля вручную

Вы знаете, что узел не возвращается, но для правильной обработки клиентов вам нужно, чтобы все три модуля работали. Вам нужно переназначить модуль `kubia-0` на здоровый узел. Как отмечалось ранее, вам нужно удалить узел или модуль вручную.

Удаление модуля обычным способом

Удалите модуль таким образом, каким вы всегда удаляли модули:

```
$ kubectl delete po kubia-0
pod "kubia-0" deleted
```

Готово, верно? При удалении модуля набор StatefulSet должен немедленно создать сменный модуль, который будет назначен на один из оставшихся узлов. Для того чтобы получить подтверждение, еще раз выведите список модулей:

```
$ kubectl get po
NAME     READY  STATUS   RESTARTS  AGE
kubia-0  1/1    Unknown  0         15m
kubia-1  1/1    Running  0         14m
kubia-2  1/1    Running  0         13m
```

Странно. Минуту назад вы удалили модуль, и `kubectl` сообщил, что он его удалил. Почему же тогда тот же модуль по-прежнему там?

> **ПРИМЕЧАНИЕ.** Модуль `kubia-0` в списке не является новым модулем с тем же именем – это ясно, глядя на столбец возраста AGE. Если бы он был новым, его возраст был бы всего несколько секунд.

Почему модуль не удаляется

Модуль был помечен на удаление еще до того, как вы его удалили. Это связано с тем, что его уже удалила сама плоскость управления (чтобы вытеснить его из узла).

Если вы снова посмотрите на листинг 10.14, то увидите, что статус модуля – `Terminating`. Модуль уже был помечен на удаление ранее и будет удален, как только Kubelet на его узле уведомит сервер API, что контейнеры модуля завершили работу. А поскольку сеть узла не работает, то этого никогда не произойдет.

Принудительное удаление модуля

Единственное, что вы можете сделать, – это сообщить серверу API удалить модуль, не дожидаясь Kubelet, который подтвердит, что модуль больше не работает. Это делается следующим образом:

```
$ kubectl delete po kubia-0 --force --grace-period 0
warning: Immediate deletion does not wait for confirmation that the running
resource has been terminated. The resource may continue to run on the
cluster indefinitely.
pod "kubia-0" deleted
```

Вам нужно использовать оба параметра `--force` и `--grace-period 0`. Предупреждение, выводимое `kubectl`, уведомляет о том, что вы сделали. Если вы снова выведете список модулей, то вы, наконец, увидите, что создан новый модуль `kubia-0`:

```
$ kubectl get po
NAME     READY  STATUS             RESTARTS  AGE
```

```
kubia-0  0/1    ContainerCreating  0         8s
kubia-1  1/1    Running            0         20m
kubia-2  1/1    Running            0         19m
```

ПРЕДУПРЕЖДЕНИЕ. Не удаляйте модули с внутренним состоянием принудительно, если вы не знаете, что узел больше не работает или недоступен (и останется в таком состоянии всегда).

Прежде чем продолжить, вам может потребоваться вернуть отключенный узел в рабочее состояние. Это можно сделать путем перезапуска узла через веб-консоль GCE либо в терминале, выполнив следующую ниже команду:

```
$ gcloud compute instances reset <имя узла>
```

10.6 Резюме

На этом завершается глава об использовании наборов модулей с внутренним состоянием StatefulSet для развертывания приложений с внутренним состоянием. В этой главе показано, как:

- предоставлять реплицированным модулям индивидуальное хранилище;
- обеспечивать стабильную идентичность модуля;
- создавать наборы StatefulSet и соответствующую управляющую службу;
- масштабировать и обновлять набор StatefulSet;
- обнаруживать других членов набора StatefulSet посредством DNS;
- подключаться к другим участникам посредством их хостнеймов;
- принудительно удалять модули с внутренним состоянием.

Теперь, когда вы знаете основные строительные блоки, которые вы можете использовать для запуска и управления приложениями Kubernetes, мы можем посмотреть более внимательно на то, как это делается. В следующей главе вы узнаете об отдельных компонентах, которые управляют кластером Kubernetes и поддерживают работу приложений.

Глава 11

Внутреннее устройство Kubernetes

Эта глава посвящена тому:

- какие компоненты составляют кластер Kubernetes;
- что делает каждый компонент и как он это делает;
- как создание объекта развертывания приводит к запуску модуля;
- что такое работающий модуль;
- как работает сеть между модулями;
- как работают службы Kubernetes;
- как достигается высокая доступность.

Прочитав данную книгу до этого места, вы ознакомились с тем, что система Kubernetes может предложить и что она делает. Но до сих пор я преднамеренно не тратил много времени на объяснение того, как именно все происходит, потому что, на мой взгляд, нет смысла вдаваться в детали работы системы до того, как у вас сформируется четкое понимание того, что система делает. Вот почему мы не говорили о том, как именно модуль назначается или как различные контроллеры, работающие внутри менеджера контроллеров, запускают развернутые ресурсы. Поскольку теперь вы знаете большинство ресурсов, которые могут быть развернуты в Kubernetes, пришло время погрузиться в то, как они реализуются.

11.1 Архитектура

Прежде чем вы обратитесь к тому, как Kubernetes делает то, что он делает, давайте более подробно рассмотрим компоненты, которые составляют кластер Kubernetes. В главе 1 вы увидели, что кластер Kubernetes делится на две части:

- плоскость управления Kubernetes;
- (рабочие) узлы.

Давайте рассмотрим подробнее, что эти две части делают и что происходит внутри них.

Компоненты плоскости управления

Плоскость управления – это то, что управляет и заставляет весь кластер функционировать. Чтобы освежить вашу память, плоскость управления состоит из следующих компонентов:

- распределенное постоянное хранилище etcd;
- сервер API;
- планировщик (Scheduler);
- менеджер контроллеров (Controller Manager).

Эти компоненты сохраняют и управляют состоянием кластера, но они не запускают контейнеры приложений.

Компоненты, работающие на рабочих узлах

Задача запуска контейнеров зависит от компонентов, работающих на каждом рабочем узле:

- агент Kubelet;
- служебный прокси Kubernetes (kube-proxy);
- среда выполнения контейнеров (Docker, rkt или др.).

Дополнительные компоненты

Помимо компонентов плоскости управления и компонентов, работающих на узлах, для обеспечения всего, что обсуждалось до сих пор, кластеру необходимо иметь несколько дополнительных компонентов. Они включают:

- DNS-сервер Kubernetes;
- панель управления (Dashboard);
- контроллер Ingress;
- компонент Heapster, о котором мы поговорим в главе 14;
- сетевой плагин контейнерного сетевого интерфейса (мы поясним, что это такое, позже в этой главе).

11.1.1 Распределенная природа компонентов Kubernetes

Все ранее упомянутые компоненты выполняются как отдельные процессы. Компоненты и их взаимосвязи показаны на рис. 11.1.

Чтобы получить все функциональные возможности, предоставляемые Kubernetes, все эти компоненты должны быть запущены. Но некоторые из них могут также выполнять полезную работу индивидуально без других компонентов. Вы увидите, как, когда мы займемся исследованием каждого из них.

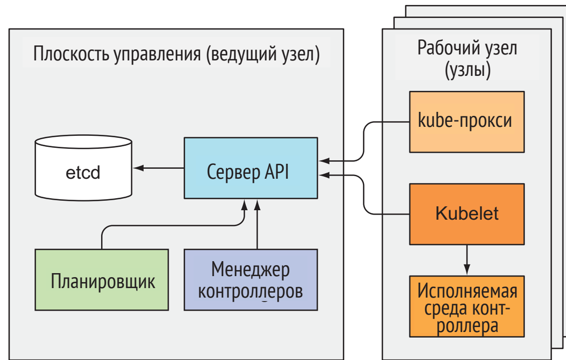


Рис. 11.1. Компоненты плоскости управления Kubernetes и рабочих узлов

Проверка статуса компонентов плоскости управления

Сервер API предоставляет ресурс API под названием ComponentStatus, который показывает состояние работоспособности каждого компонента плоскости управления. Список компонентов и их статусов можно вывести с помощью `kubectl`:

```
$ kubectl get componentstatuses
NAME                STATUS   MESSAGE             ERROR
Scheduler           Healthy  ok
controller-manager  Healthy  ok
etcd-0              Healthy  {"health": "true"}
```

Как эти компоненты взаимодействуют

Компоненты системы Kubernetes взаимодействуют только с сервером API. Они не обмениваются друг с другом напрямую. Сервер API является единственным компонентом, который взаимодействует с хранилищем etcd. Ни один из других компонентов не взаимодействует с хранилищем etcd напрямую. Вместо этого они модифицируют состояние кластера, обмениваясь с сервером API.

Подключения между сервером API и другими компонентами почти всегда инициируются компонентами, как показано на рис. 11.1. Но сервер API подключается к агенту Kubelet, когда вы используете `kubectl` для выборки журналов, используете команду `kubectl attach` для подключения к запущенному контейнеру или используете команду `kubectl port-forward`.

ПРИМЕЧАНИЕ. Команда `kubectl attach` аналогична команде `kubectl exec`, но закрепляется за главным процессом, запущенным в контейнере, а не за дополнительным процессом.

Запуск нескольких экземпляров отдельных компонентов

Хотя все компоненты на рабочих узлах должны выполняться на одном узле, компоненты плоскости управления могут быть легко распределены по нескольким серверам. Для обеспечения высокой доступности может быть запущено несколько экземпляров каждого компонента плоскости управления. Несмотря на то что одновременно может быть активно несколько экземпляров хранилища etcd и сервера API и они в действительности могут выполнять свои задания параллельно, в конкретный момент времени может быть активен только один экземпляр планировщика и менеджера контроллеров – при этом другие находятся в режиме ожидания.

Как выполняются компоненты

Компоненты плоскости управления, а также kube-proxy могут быть развернуты в системе напрямую или работать как модули (как показано в листинге 11.1). Вы можете удивиться, услышав это, но все это будет иметь смысл позже, когда мы поговорим об агенте Kubelet.

Kubelet – это единственный компонент, который всегда выполняется как обычный системный компонент, и именно Kubelet запускает все остальные компоненты как модули. Для запуска компонентов плоскости управления как модулей агент Kubelet также разворачивается на ведущем узле. В следующем ниже листинге показаны модули в пространстве имен `kube-system` в кластере, созданном с помощью инструмента `kubeadm`, подробное описание которого приводится в приложении B.

Листинг 11.1. Компоненты Kubernetes, работающие как модули

```
$ kubectl get po -o custom-columns=POD:metadata.name,NODE:spec.nodeName
➥  --sort-by spec.nodeName -n kube-system
POD                              NODE
kube-controller-manager-master   master
kube-dns-2334855451-37d9k        master
etcd-master                      master
kube-apiserver-master            master
kube-scheduler-master            master
kube-flannel-ds-tgj9k            node1
kube-proxy-ny3xm                 node1
kube-flannel-ds-0eek8            node2
kube-proxy-sp362                 node2
kube-flannel-ds-r5yf4            node3
kube-proxy-og9ac                 node3
```

etcd, сервер API, планировщик, менеджер контроллеров и DNS-сервер работают на ведущем узле

Каждый из трех узлов выполняет модуль Kube-прокси и модуль сетевого взаимодействия Flannel

Как видно из листинга, все компоненты плоскости управления работают как модули на ведущем узле. Имеется три рабочих узла, и каждый из них выполняет модуль kube-proxy и модуль Flannel, который предоставляет для модулей оверлейную сеть (мы поговорим о надстройке Flannel позже).

СОВЕТ. Как показано в листинге, с помощью параметра `-o custom-columns` можно поручить `kubectl` показывать настраиваемые столбцы, а с помощью параметра `--sort-by` – отсортировать список ресурсов.

Теперь давайте рассмотрим каждый из компонентов вблизи, начиная с самого низкоуровневого компонента плоскости управления – постоянного хранилища.

11.1.2 Как Kubernetes использует хранилище etcd

Все объекты, которые вы создавали в этой книге, – модули, контроллеры репликации, службы, секреты и т. д. – должны храниться где-то постоянно, чтобы их манифесты выдерживали перезапуски и аварийные сбои сервера API. Для этого Kubernetes использует хранилище etcd, которое представляет собой быстрое, распределенное и согласованное хранилище в формате ключ-значение. Поскольку хранилище etcd является распределенным, для обеспечения высокой доступности и повышения производительности вы можете запускать несколько его экземпляров.

Единственным компонентом, который напрямую взаимодействует с хранилищем etcd, является сервер API Kubernetes. Все остальные компоненты читают и записывают данные в хранилище etcd косвенно через сервер API. Это приносит немного преимуществ, среди которых более устойчивая система оптимистической блокировки, а также валидация; и, абстрагируя фактический механизм хранения от всех других компонентов, его гораздо проще заменить в будущем. Стоит подчеркнуть, что хранилище etcd является *единственным* местом, где Kubernetes хранит состояние кластера и метаданные.

Об оптимистическом управлении параллелизмом

Оптимистическое управление параллелизмом (иногда называемое оптимистической блокировкой) – это метод, в котором вместо блокировки порции данных и предотвращения ее чтения или обновления порция данных во время блокировки содержит номер версии. При каждом обновлении данных номер версии увеличивается. При обновлении данных проверяется, увеличился ли номер версии между временем чтения данных клиентом и временем отправки им обновления. Если это происходит, то обновление отклоняется, и клиент должен повторно прочитать новые данные и попытаться обновить их снова.

В результате этого, когда два клиента пытаются обновить одну и ту же запись данных, успешно выполняется только первая.

Все ресурсы Kubernetes содержат поле `metadata.resourceVersion`, которое клиенты должны передавать обратно на сервер API при обновлении объекта. Если эта версия не совпадает с версией, хранящейся в etcd, то сервер API обновление отклоняет.

Как хранятся ресурсы в хранилище etcd

Во время написания книги Kubernetes может использовать либо etcd версии 2, либо версии 3, но версия 3 теперь рекомендуется вследствие повышенной производительности. В хранилище etcd v2 ключи хранятся в иерархическом пространстве ключей, что делает пары ключ-значение похожими на файлы в файловой системе. Каждый ключ в etcd является либо каталогом, который содержит другие ключи, либо обычным ключом с соответствующим значением. В хранилище etcd v3 каталоги не поддерживаются, но поскольку формат ключа остается прежним (ключи могут включать слеши), вы все равно можете думать о них как о сгруппированных в каталоги. Kubernetes хранит все свои данные в etcd в /registry. Ниже приведен список ключей, находящихся в /registry.

Листинг 11.2. Записи верхнего уровня, хранимые в etcd системой Kubernetes

```
$ etcdctl ls /registry
/registry/configmaps
/registry/daemonsets
/registry/deployments
/registry/events
/registry/namespaces
/registry/pods
...
```

Вы поймете, что эти ключи соответствуют типам ресурсов, о которых вы узнали в предыдущих главах.

> **ПРИМЕЧАНИЕ.** Если вы используете v3 API etcd, то для просмотра содержимого каталога вы не сможете использовать команду `ls`. Вместо этого вы можете вывести список всех ключей, которые начинаются с заданного префикса. Для этого используется команда `etcdctl get / registry --prefix=true`.

В следующем ниже листинге показано содержимое каталога /registry/pods.

Листинг 11.3. Ключи в каталоге /registry/pods

```
$ etcdctl ls /registry/pods
/registry/pods/default
/registry/pods/kube-system
```

Как можно сделать вывод из имен, эти две записи соответствуют пространствам имен `default` и `kube-system`. Это означает, что модули хранятся в соответствии с пространством имен. В следующем ниже листинге показаны записи в каталоге /registry/pods/default.

Листинг 11.4. Записи хранилища etcd для модулей в пространстве имен default

```
$ etcdctl ls /registry/pods/default
/registry/pods/default/kubia-159041347-xk0vc
/registry/pods/default/kubia-159041347-wt6ga
/registry/pods/default/kubia-159041347-hp2o5
```

Каждая запись соответствует отдельному модулю. Это не каталоги, а записи типа «ключ-значение». В следующем ниже листинге показано, что хранится в одном из них.

Листинг 11.5. Запись хранилища etcd, представляющая модуль

```
$ etcdctl get /registry/pods/default/kubia-159041347-wt6ga
{"kind":"Pod","apiVersion":"v1","metadata":{"name":"kubia-159041347-wt6ga",
"generateName":"kubia-159041347-","namespace":"default","selfLink":...
```

Вы сразу поймете, что это не что иное, как определение модуля в формате JSON. В хранилище etcd сервер API хранит полное представление ресурса в формате JSON. Из-за иерархического пространства ключей хранилища etcd все хранимые ресурсы можно рассматривать как файлы JSON в файловой системе. Просто, не правда ли?

> **ПРЕДУПРЕЖДЕНИЕ.** До Kubernetes версии 1.7 манифест JSON ресурса Secret тоже хранился таким образом (он не был зашифрован). Если кто-то получал прямой доступ к хранилищу etcd, он знал все ваши секреты. Начиная с версии 1.7 секреты шифруются и, следовательно, хранятся гораздо надежнее.

Обеспечение согласованности и достоверности хранимых объектов

Помните про системы Borg и Omega компании Google, упомянутые в главе 1, на которых основана система Kubernetes? Как и Kubernetes, система Omega для хранения состояния кластера тоже использует централизованное хранилище, но, в отличие от Kubernetes, несколько компонентов плоскости управления обращается к хранилищу напрямую. Все эти компоненты должны быть уверены, что все они придерживаются одного и того же оптимистического механизма блокировки с целью правильной обработки конфликтов. Один компонент, не полностью поддерживающий этот механизм, может привести к несогласованности данных.

Система Kubernetes этот подход совершенствует, требуя, чтобы все другие компоненты плоскости управления проходили через сервер API. Благодаря этому все обновления состояния кластера всегда согласованы, поскольку механизм оптимистической блокировки реализован в одном месте, и поэтому существует меньше шансов на ошибку, если вообще такой шанс существует. Сервер API также гарантирует, что данные, записанные в хранилище, всегда

действительны и что изменения данных выполняются только авторизованными клиентами.

Обеспечение согласованности, когда хранилище etcd кластеризовано

Для обеспечения высокой доступности обычно выполняется более одного экземпляра хранилища etcd. Множественные экземпляры etcd должны оставаться согласованными. Такая распределенная система должна прийти к консенсусу относительно того, каким является фактическое состояние. Для достижения этого в хранилище etcd используется консенсусный алгоритм RAFT, который гарантирует, что в любой момент состояние каждого узла является либо тем, что большинство узлов соглашается считать текущим состоянием, либо одним из ранее согласованных состояний.

Клиенты, подключающиеся к различным узлам кластера etcd, будут видеть либо фактическое текущее состояние, либо одно из состояний из прошлого (в Kubernetes единственным клиентом etcd является сервер API, но может быть несколько экземпляров).

Для перехода кластера в следующее состояние консенсусному алгоритму требуется большинство (или кворум). В результате этого, если кластер разделится на две несвязанные группы узлов, состояние в этих двух группах никогда не может расходиться, так как для перехода из предыдущего состояния в новое требуется более половины узлов, принимающих участие в изменении состояния. Если одна группа содержит большинство всех узлов, то другая, очевидно, его не содержит. Первая группа может изменять состояние кластера, а другая – нет. Когда две группы переподключаются, вторая группа может догнать состояние в первой группе (см. рис. 11.2).

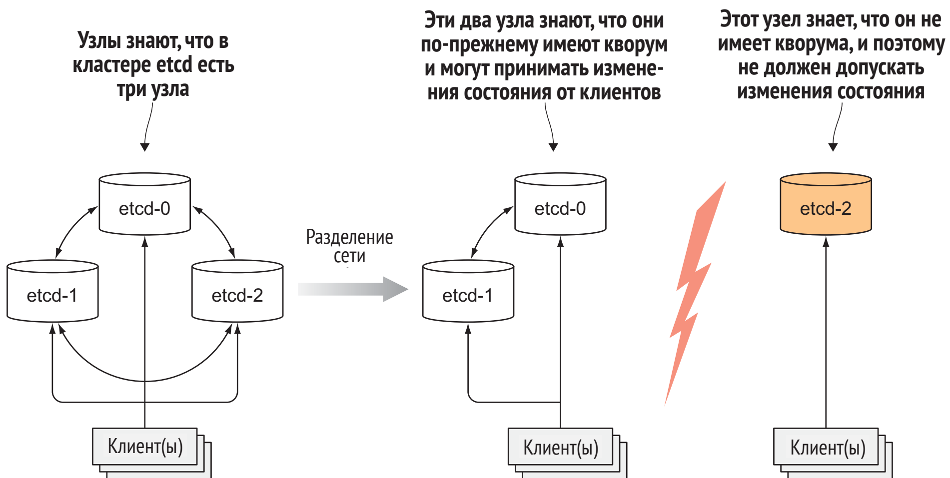


Рис. 11.2. В сценарии с разделением консенсуса изменения состояния принимает только та сторона, которая по-прежнему обладает большинством (кворумом)

Почему количество экземпляров хранилища etcd должно быть нечетным

Хранилище etcd обычно развертывается с нечетным числом экземпляров. Уверен, что вы хотели бы знать, почему. Давайте разберем наличие двух экземпляров и одного. Наличие двух экземпляров требует, чтобы для обеспечения большинства было оба экземпляра. Если любой из них завершает работу аварийно, то кластер etcd не может перейти в новое состояние, так как нет большинства. Наличие двух экземпляров хуже, чем наличие всего одного экземпляра. Имея два, вероятность аварийного сбоя всего кластера увеличилась на 100%, по сравнению с аварийным сбоем одноузлового кластера.

То же самое касается сравнения трех и четырех экземпляров etcd. В случае с тремя экземплярами один экземпляр может завершить работу аварийно, и большинство (состоящее из двух) по-прежнему существует. В случае с четырьмя экземплярами для обеспечения большинства вам нужны три узла (двух недостаточно). В кластерах с тремя и четырьмя экземплярами может произойти аварийный сбой всего одного экземпляра. Но при выполнении четырех экземпляров, если один из них завершает работу аварийно, существует более высокая вероятность того, что дополнительный экземпляр из трех оставшихся экземпляров завершит работу аварийно (по сравнению с трехузловым кластером с одним аварийным узлом и двумя оставшимися узлами).

Обычно для более крупных кластеров достаточно кластера etcd из пяти или семи узлов. Он может справляться соответственно с двух- или трехузловым аварийным сбоем, что достаточно почти во всех ситуациях.

11.1.3 Что делает сервер API

Сервер API Kubernetes является центральным компонентом, используемым всеми другими компонентами и клиентами, такими как `kubectl`. Он предоставляет интерфейс CRUD (Create, Read, Update, Delete) для запросов и изменения состояния кластера через API RESTful. Он хранит это состояние в хранилище etcd.

В дополнение к обеспечению согласованного способа хранения объектов в хранилище etcd он также выполняет валидацию этих объектов, так что клиенты не могут хранить в нем неправильно сконфигурированные объекты (что они могли бы, если бы они писали в хранилище напрямую). Наряду с валидацией он также занимается оптимистической блокировкой, поэтому в случае одновременных обновлений изменения объекта никогда не переопределяются другими клиентами.

Одним из клиентов сервера API является инструмент командной строки `kubectl`, который вы использовали с самого начала книги. Например, при создании ресурса из файла JSON инструмент `kubectl` отправляет содержимое файла на сервер API посредством запроса HTTP POST. На рис. 11.3 показано, что происходит внутри сервера API при получении запроса. Это более подробно разъясняется в следующих нескольких абзацах.

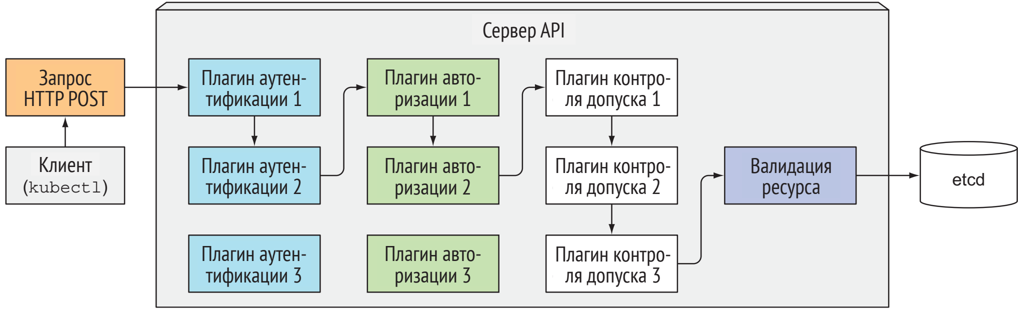


Рис. 11.3. Работа сервера API

Аутентификация клиента с помощью плагинов аутентификации

Прежде всего сервер API должен аутентифицировать клиента, отправляющего запрос. Это осуществляется с помощью одного или нескольких плагинов аутентификации на сервере API. Сервер API вызывает эти плагины по очереди до тех пор, пока один из них не определит, кто отправляет запрос. Это делается путем инспектирования HTTP-запроса.

В зависимости от метода аутентификации пользователь может быть извлечен из сертификата клиента или заголовка HTTP, к примеру `Authorization`, который вы применяли в главе 8. Плагин извлекает имя пользователя, идентификатор пользователя, а также группы, к которым пользователь принадлежит. Эти данные затем используются на следующем этапе, этапе авторизации.

Авторизация клиента с помощью плагинов авторизации

Помимо плагинов аутентификации, сервер API также настроен на использование одного или нескольких плагинов авторизации. Их задача – определять, может ли прошедший проверку пользователь выполнять запрошенное действие над запрошенным ресурсом. Например, при создании модулей сервер API по очереди консультируется со всеми плагинами авторизации, чтобы определить, может ли пользователь создавать модули в запрошенном пространстве имен. Как только плагин сообщает, что пользователь может выполнить действие, сервер API переходит к следующему этапу.

Валидация и/или модификация ресурса в запросе с помощью плагинов контроля допуска

Если запрос пытается создать, изменить или удалить ресурс, запрос отправляется через контроль допуска. Опять же, сервер сконфигурирован с несколькими плагинами контроля допуска. Эти плагины могут изменять ресурс по разным причинам. Они могут инициализировать поля, отсутствующие в спецификации ресурса, в сконфигурированные значения по умолчанию или даже переопределять их. Они могут даже изменять другие родственные ресурсы, которых нет в запросе, а также могут отклонять запрос по любой причине. Ресурс проходит через все плагины контроля допуска.

ПРИМЕЧАНИЕ. Когда запрос пытается всего лишь прочитать данные, запрос через контроль допуска не проходит.

Примеры плагинов контроля допуска включают:

- `AlwaysPullImages` – переопределяет политику `imagePullPolicy` модуля, присваивая ей значение `Always` и заставляя извлекать образ всякий раз, когда модуль развертывается;
- `ServiceAccount` – применяет принятую по умолчанию учетную запись службы к модулям, которые не задают ее явно;
- `NamespaceLifecycle` – предотвращает создание модулей в пространствах имен, которые находятся в процессе удаления, а также в несуществующих пространствах имен;
- `ResourceQuota` – гарантирует, что модули в определенном пространстве имен используют только такой объем ЦП и памяти, который был выделен пространству имен. Мы узнаем об этом больше в главе 14.

Вы найдете список дополнительных плагинов контроля допуска в документации Kubernetes по адресу https://kubernetes.io/docs/admin/admission-controllers/.

Валидация ресурса и его постоянное хранение

После прохождения запроса через все плагины контроля допуска сервер API выполняет валидацию объекта, сохраняет его в хранилище etcd и возвращает отклик клиенту.

11.1.4 Как сервер API уведомляет клиентов об изменениях ресурсов

Сервер API ничего не делает, кроме того что мы обсуждали. Например, он не создает модули при создании ресурса ReplicaSet и не управляет конечными точками службы. Этим занимаются контроллеры в менеджере контроллеров.

Но сервер API даже не говорит этим контроллерам, что делать. Он лишь позволяет этим контроллерам и другим компонентам наблюдать за изменениями в развернутых ресурсах. Компонент плоскости управления может запросить уведомление при создании, изменении или удалении ресурса. Это позволяет компоненту выполнять любую задачу, необходимую в ответ на изменение метаданных кластера.

Клиенты следят за изменениями, открывая соединение HTTP с сервером API. Через это соединение клиент будет получать поток изменений в наблюдаемых объектах. При каждом обновлении объекта сервер отправляет новую версию объекта всем подключенным клиентам, наблюдающим за объектом. Рисунок 11.4 показывает, как клиенты могут наблюдать за изменениями в модулях и как изменение в одном из модулей сохраняется в хранилище etcd и затем уведомление об изменении передается всем клиентам, наблюдающим за модулями в тот момент.

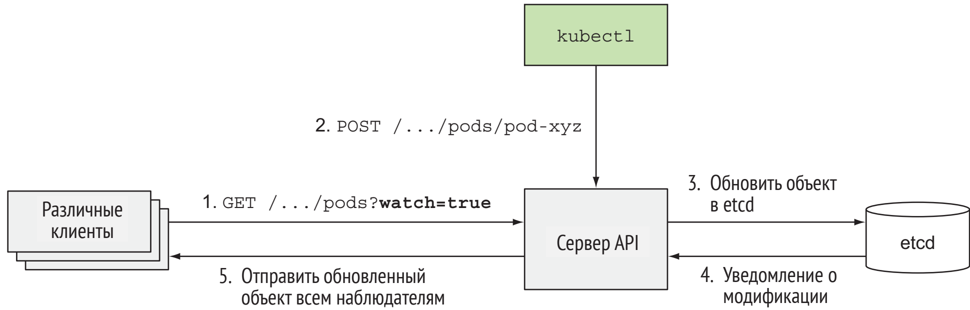


Рис. 11.4. При обновлении объекта сервер API отправляет обновленный объект всем заинтересованным наблюдателям

Одним из клиентов сервера API является инструмент `kubectl`, который также поддерживает просмотр ресурсов. Например, при развертывании модуля не нужно постоянно опрашивать список модулей, повторно выполняя команду `kubectl get pod`. Вместо этого можно использовать флаг `--watch` и получать уведомления о каждом создании, изменении или удалении модуля, как показано в следующем ниже листинге.

Листинг 11.6. Просмотр создаваемого и удаляемого модуля

```
$ kubectl get pods --watch
NAME                    READY   STATUS              RESTARTS   AGE
kubia-159041347-14j3i   0/1     Pending             0          0s
kubia-159041347-14j3i   0/1     Pending             0          0s
kubia-159041347-14j3i   0/1     ContainerCreating   0          1s
kubia-159041347-14j3i   0/1     Running             0          3s
kubia-159041347-14j3i   1/1     Running             0          5s
kubia-159041347-14j3i   1/1     Terminating         0          9s
kubia-159041347-14j3i   0/1     Terminating         0          17s
kubia-159041347-14j3i   0/1     Terminating         0          17s
kubia-159041347-14j3i   0/1     Terminating         0          17s
```

Вы даже можете заставить `kubectl` распечатывать весь YAML на каждом событии наблюдения, как показано ниже:

```
$ kubectl get pods -o yaml --watch
```

Механизм наблюдения также используется планировщиком, являющимся следующим компонентом плоскости управления, о котором вы узнаете больше.

11.1.5 Планировщик

Вы уже знаете, что вы обычно не указываете, на каком узле кластера должен работать модуль. Эта работа оставлена на усмотрение планировщика. Издалека работа планировщика выглядит простой. Вся его работа заключается в том,

чтобы на основе реализованного в сервере API механизма наблюдения ждать вновь созданных модулей и назначать узел для каждого нового модуля, для которого узел еще не был задан.

Планировщик не предписывает выбранному узлу (или агенту Kubelet, работающему на этом узле) запускать модуль. Планировщик лишь обновляет определение модуля через сервер API. Затем сервер API уведомляет Kubelet (опять же, через механизм наблюдения, описанный ранее) о том, что модуль назначен. Как только агент Kubelet на целевом узле увидит, что модуль назначен на его узел, он создает и запускает контейнеры модуля.

Хотя в крупном плане вид данного процесса назначения модулю узла кажется тривиальным, фактическая задача выбора самого лучшего узла для модуля не так проста. Разумеется, самый простой планировщик может выбирать случайный узел и не заботиться о модулях, уже работающих на этом узле. С другой стороны, планировщик может использовать передовые методы, такие как машинное обучение, чтобы предвидеть, какие модули должны быть назначены в ближайшие минуты или часы, и назначать модули с целью максимизировать предстоящую задействованность оборудования, не требуя никакого переназначения существующих модулей. Используемый по умолчанию планировщик Kubernetes находится где-то посередине.

Принятый по умолчанию алгоритм назначения

Как показано на рис. 11.5, процесс выбора узла может быть разбит на две части:

- фильтрация списка всех узлов для получения списка приемлемых узлов, на которые можно назначить модуль;
- приоритизация приемлемых узлов и выбор самого лучшего. Если несколько узлов имеют самый высокий балл, используется циклический перебор, чтобы гарантировать, что модули развернуты по всем из них равномерно.

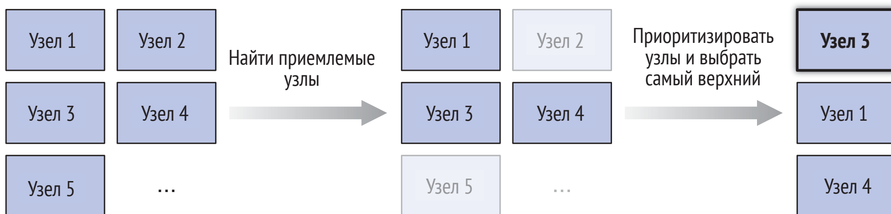


Рис. 11.5. Планировщик находит приемлемые узлы для модуля и затем выбирает для модуля самый лучший узел

Поиск приемлемых узлов

Чтобы определить, какие узлы приемлемы для модуля, планировщик пропускает каждый узел через список сконфигурированных предикативных функций. Они проверяют различные аспекты, такие как:

- может ли узел выполнять запросы модуля для аппаратных ресурсов? Вы узнаете, как их задавать, в главе 14;
- заканчиваются ли в узле ресурсы (сообщает ли он о состоянии дефицита памяти или дискового пространства);
- если модуль запрашивает назначения на конкретный узел (по имени), то является ли узел тем самым узлом;
- имеет ли узел метку, которая совпадает с селектором узлов в спецификации модуля (если он определен);
- если модуль запрашивает привязку к конкретному порту хоста (обсуждается в главе 13), принят ли этот порт уже на этом узле или нет;
- если модуль запрашивает конкретный тип тома, может ли этот том быть смонтирован для этого модуля на этом узле или же другой модуль на узле уже использует тот же том;
- допускает ли модуль ограничения узла? Ограничения и допуски описаны в главе 16;
- указывает ли модуль на правила узлового и/или модульного сходства или антисходства? Если да, то будет ли назначение модуля на этот узел нарушать эти правила? Это также объясняется в главе 16.

Все эти проверки должны успешно пройти, для того чтобы узел имел право на размещение модуля. После выполнения этих проверок на каждом узле планировщик получает подмножество узлов. Любой из этих узлов может запустить модуль, поскольку у них достаточно доступных ресурсов для модуля и они соответствуют всем требованиям, указанным в определении модуля.

Выбор самого лучшего узла для модуля

Несмотря на то что все эти узлы приемлемы и могут запускать модуль, некоторые из них могут быть лучше, чем другие. Предположим, у вас есть кластер из двух узлов. Оба узла имеют право, но один уже управляет 10 модулями, в то время как другой прямо сейчас по какой-то причине никакими модулями не управляет. Очевидно, что планировщик в этом случае должен отдавать предпочтение второму узлу.

Или нет? Если эти два узла предоставлены облачной инфраструктурой, может случиться так, что лучше назначить модуль первому узлу и отказаться от права на второй узел, вернув его облачному провайдеру, чтобы сэкономить деньги.

Продвинутое назначение модулей

Рассмотрим еще один пример. Представьте, что у вас есть несколько реплик модуля. В идеале, вы хотите, чтобы они распределялись по как можно большему количеству узлов, вместо того чтобы все они были приписаны к одному. Аварийный сбой этого узла приведет к недоступности службы, поддерживаемой этими модулями. Но если модули были распределены по разным узлам, аварийный сбой одного узла едва оставит даже царапину на емкости службы.

Модули, принадлежащие той же службе или набору реплик ReplicaSet, по умолчанию распределены по нескольким узлам. Вместе с тем не гарантируется, что это всегда так. Но вы можете заставить модули распределяться по кластеру или держаться вместе, определив правила сходства и антисходства модулей, которые описаны в главе 16.

Даже эти два простых случая показывают, насколько сложным может быть процесс назначения модулей узлам, поскольку он зависит от множества факторов. По этой причине планировщик может быть либо настроен в соответствии с вашими конкретными потребностями или спецификой инфраструктуры, либо даже быть заменен самостоятельной реализацией. Вы также можете запустить кластер Kubernetes без планировщика, но тогда вам придется выполнять назначение вручную.

Использование нескольких планировщиков

Вместо выполнения одного планировщика в кластере можно выполнять несколько планировщиков. Затем для каждого модуля указывать планировщика, который должен назначать конкретный модуль, задав в спецификации модуля свойство `schedulerName`.

Модули без этого набора свойств назначаются с помощью планировщика, установленного по умолчанию, и то же касается модулей со значением `schedulerName`, равным `default-scheduler`. Все остальные модули планировщиком по умолчанию игнорируются, поэтому они должны назначаться вручную или другим планировщиком, наблюдающим за такими модулями.

Вы можете реализовать собственных планировщиков и разворачивать их в кластере или разворачивать дополнительный экземпляр планировщика Kubernetes с разными параметрами конфигурации.

11.1.6 Знакомство с контроллерами, работающими в менеджере контроллеров

Как отмечалось ранее, сервер API ничего не делает, кроме того что хранит ресурсы в хранилище etcd и уведомляет клиентов об изменении. Планировщик только назначает узел модулю, поэтому, для того чтобы убедиться, что фактическое состояние системы сходится к желаемому состоянию, как указано в ресурсах, развернутых через сервер API, вам нужны другие активные компоненты. Эта работа выполняется контроллерами, работающими в менеджере контроллеров.

Единый процесс менеджера контроллеров в настоящее время объединяет множество контроллеров, выполняющих различные задачи согласования. В конечном итоге эти контроллеры будут разделены на отдельные процессы, что позволит при необходимости заменять каждый из них собственной реализацией. Список этих контроллеров включает:

- контроллер репликации (контроллер для ресурсов ReplicationController);
- контроллер набора реплик ReplicaSet, набора демонов DaemonSet и задания Job;

- контроллер ресурса развертывания Deployment;
- контроллер набора модулей с внутренним состоянием StatefulSet;
- контроллер узла;
- контроллер службы Service;
- контроллер конечных точек Endpoints;
- контроллер пространства имен Namespace;
- контроллер постоянного тома PersistentVolume;
- другие.

То, что делает каждый из этих контроллеров, должно быть видно из его имени. По этому списку вы можете сказать, что почти для каждого ресурса есть свой контроллер, который вы можете создать. Ресурсы – это описания того, что должно выполняться в кластере, тогда как контроллеры – это активные компоненты Kubernetes, которые выполняют фактическую работу в результате развертывания ресурсов.

Несколько подсказок по поводу того, как исследовать исходный код контроллеров

Если вы заинтересованы в том, как именно работают эти контроллеры, настоятельно рекомендуется просмотреть их исходный код. Чтобы было проще, вот несколько советов:

Исходный код для контроллеров доступен по адресу https://github.com/kubernetes/kubernetes/blob/master/pkg/controller.

Каждый контроллер обычно имеет конструктор, в котором он создает объект-информатор `Informer`. Это, в сущности, слушающий объект, который вызывается всякий раз, когда объект API обновляется. Обычно информер прослушивает изменения в ресурсе определенного типа. Исследовав конструктор, вы увидите, за какими ресурсами контроллер наблюдает.

Затем перейдите к методу `worker()`. В нем вы найдете метод, который вызывается всякий раз, когда контроллер должен что-то сделать. Фактическая функция часто хранится в поле `syncHandler` или в чем-то подобном. Это поле также инициализируется в конструкторе, поэтому здесь вы найдете имя вызываемой функции. Эта функция является тем самым местом, где происходит все волшебство.

Что делают контроллеры, и как они это делают

Контроллеры делают много разных вещей, но все они наблюдают за изменениями ресурсов (развертываниями, службами и т. д.) на сервере API и выполняют операции для каждого изменения, будь то создание нового объекта или обновление или удаление существующего объекта. В большинстве случаев эти операции включают создание других ресурсов или обновление самих отслеживаемых ресурсов (например, для обновления состояния объекта).

Как правило, контроллеры выполняют цикл согласования, который согласовывает фактическое состояние с требуемым состоянием (указанным в секции `spec` ресурса) и записывают новое фактическое состояние в секцию `status` ресурса. Для того чтобы получать уведомления об изменениях, контроллеры используют механизм наблюдения, но поскольку использование наблюдений не гарантирует, что контроллер не пропустит событие, они также периодически выполняют операцию запроса списка, чтобы убедиться, что они ничего не пропустили.

Контроллеры никогда не обмениваются друг с другом напрямую. Они даже не знают, что существуют другие контроллеры. Каждый контроллер подключается к серверу API через механизм наблюдения, описанный в разделе 11.1.3, запрашивает об уведомлении, когда в списке ресурсов любого типа, за который отвечает контроллер, происходит изменение.

Мы кратко рассмотрим, что делает каждый из контроллеров, но если вы хотите получить подробное представление о том, что они делают, я предлагаю вам взглянуть на их исходный код напрямую. Приведенная ниже вставка объясняет, как начать работу.

Менеджер репликации

Контроллер, который оживляет ресурсы ReplicationController, называется менеджером репликации. Мы говорили о том, как работают контроллеры репликации, в главе 4. Фактическую работу делают не контроллеры репликации, а менеджер репликации. Давайте вкратце рассмотрим, что делает контроллер, потому что это поможет вам понять остальные контроллеры.

В главе 4 мы отметили, что операция контроллера репликации может рассматриваться как бесконечный цикл, где в каждой итерации контроллер находит количество модулей, соответствующих его селектору модулей, и сравнивает их количество с требуемым количеством реплик.

Теперь, когда вы знаете, как сервер API может уведомлять клиентов через механизм наблюдения, ясно, что контроллер не опрашивает модули в каждой итерации, а вместо этого уведомляется механизмом наблюдения о каждом изменении, которое может повлиять на требуемое количество реплик или количество совпавших модулей (см. рис. 11.6). Любые такие изменения инициируют контроллер перепроверять требуемое количество с фактическим количеством реплик и действовать соответственно.

Вы уже знаете, что когда выполняется слишком мало экземпляров модуля, контроллер репликации запускает дополнительные экземпляры. Но на самом деле он не управляет ими сам. Он создает новые манифесты модулей, отправляет их на сервер API и поручает планировщику и агенту Kubelet выполнить свою работу по назначению модуля узлу и запуску модуля.

Менеджер репликации выполняет свою работу, управляя объектами API модулей через сервер API. Так работают все контроллеры.

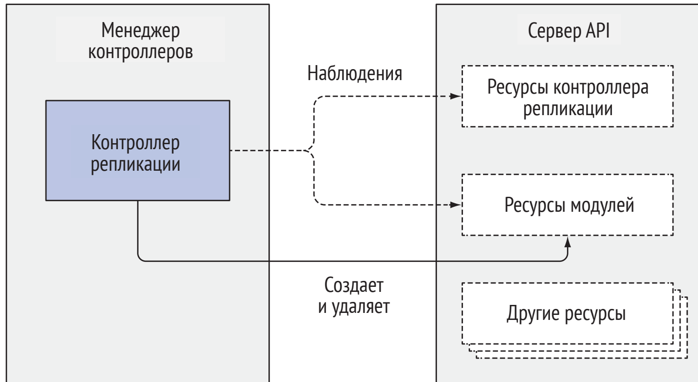


Рис. 11.6. Менеджер репликации отслеживает изменения объектов API

Контроллеры набора реплик, набора демонов и задания

Контроллер набора реплик ReplicaSet делает почти то же самое, что и менеджер репликации, описанный ранее, поэтому нам не нужно здесь чего-то добавлять. Контроллеры DaemonSet и Job похожи. Они создают ресурсы модулей из шаблона модуля, определенного в соответствующих ресурсах. Как и менеджер репликации, эти контроллеры не запускают модули, а отправляют определения модулей на сервер API, поручая агенту Kubelet создавать и запускать свои контейнеры.

Контроллер развертывания

Контроллер развертывания Deployment обеспечивает синхронизацию фактического состояния развертывания с требуемым состоянием, указанным в соответствующем объекте API Deployment.

Контроллер развертывания Deployment выполняет развертывание новой версии каждый раз при изменении объекта развертывания (если изменение должно повлиять на развернутые модули). Это делается путем создания реплик, а затем соответствующего масштабирования как старой, так и новой реплик на основе стратегии, указанной в развертывании. Масштабирование выполняется до тех пор, пока все старые модули не будут заменены новыми. Оно не создает никаких модулей напрямую.

Контроллер набора модулей с внутренним состоянием

Контроллер набора StatefulSet, подобно контроллеру набора ReplicaSet и другим родственным контроллерам, создает, управляет и удаляет модули в соответствии со спецификацией ресурса StatefulSet. Но, в отличие от других контроллеров, которые лишь управляют модулями, контроллер набора StatefulSet также создает экземпляры и управляет заявками PersistentVolumeClaim для каждого экземпляра модуля.

Контроллер узла

Контроллер узла управляет ресурсами узла, которые описывают рабочие узлы кластера. Помимо прочего, контроллер узла синхронизирует список объектов Node с фактическим списком машин, работающих в кластере. Он также отслеживает работоспособность каждого узла и удаляет модули из недоступных узлов.

Контроллер узла не является единственным компонентом, вносящим изменения в объекты Node. Они также изменяются посредством агента Kubelet и, безусловно, могут также изменяться пользователями через вызовы API REST.

Контроллер службы

В главе 5, когда мы говорили о службах, вы узнали, что существует несколько различных типов служб. Одной из них была служба `LoadBalancer`, которая запрашивает из инфраструктуры подсистему балансировки нагрузки, чтобы сделать службу доступной извне. Контроллер службы запрашивает и освобождает подсистему балансировки нагрузки из инфраструктуры при создании или удалении службы типа `LoadBalancer`.

Контроллер конечных точек

Вы помните, что службы не связаны непосредственно с модулями, а содержат список конечных точек (IP-адресов и портов), который создается и обновляется вручную или автоматически в соответствии с селектором модулей, определенным в службе. Контроллер конечных точек Endpoints является активным компонентом, который постоянно обновляет список конечных точек IP-адресами и портами модулей, соответствующих селектору меток.

Как показывает рис. 11.7, этот контроллер наблюдает и за службами, и за модулями. Когда службы добавляются или обновляются, или модули добавляются, обновляются или удаляются, он выбирает модули, соответствующие селектору модуля службы, и добавляет их IP-адреса и порты в ресурс конечных точек. Напомним, что объект Endpoints является автономным объектом, поэтому контроллер создает его по мере необходимости. Кроме того, он также удаляет объект конечных точек при удалении службы.

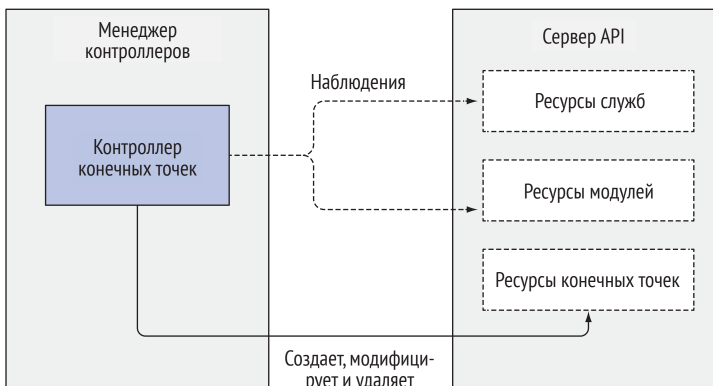


Рис. 11.7. Контроллер конечных точек наблюдает за ресурсами служб и модулей и управляет конечными точками

Контроллер пространства имен

Помните пространства имен (мы говорили о них в главе 3)? Большинство ресурсов принадлежит конкретному пространству имен. При удалении ресурса пространства имен Namespace все ресурсы в этом пространстве имен должны быть также удалены. Это то, что делает контроллер пространства имен. Когда он получает уведомление об удалении объекта Namespace, он удаляет все ресурсы, принадлежащие пространству имен, через сервер API.

Контроллер постоянного тома

В главе 6 вы узнали о постоянных томах PersistentVolume и заявках на получение постоянного тома PersistentVolumeClaim. Как только пользователь создает заявку PersistentVolumeClaim, система Kubernetes должна найти соответствующий постоянный том PersistentVolume и привязать его к заявке. Это выполняется контроллером постоянного тома.

Когда возникает заявка PersistentVolumeClaim, контроллер находит наилучшее соответствие для заявки, выбирая наименьший постоянный том с режимом доступа, соответствующим тому, который запрошен в заявке и объявленной емкости выше емкости, запрошенной в заявке. Он делает это, сохраняя упорядоченный список постоянных томов для каждого режима доступа по возрастанию емкости и возвращая первый том из списка.

Затем, когда пользователь удаляет заявку PersistentVolumeClaim, том отсоединяется и освобождается в соответствии с политикой освобождения тома (остается как есть, удаляется или очищается).

Подведение итогов относительно контроллеров

Теперь вы должны хорошо чувствовать, что делает каждый контроллер и как работают контроллеры в целом. Опять же, все эти контроллеры работают с объектами API через сервер API. Они не общаются напрямую с агентами Kubelet и не дают им никаких инструкций. На самом деле они даже не знают о существовании агентов Kubelet. После того как контроллер обновляет ресурс на сервере API, служебные прокси агентов Kubelet и системы Kubernetes, которые также в неведении о существовании контроллеров, выполняют свою работу, например разворачивают контейнеры модуля и закрепляют за ними сетевое хранилище или, в случае служб, настраивают фактическую балансировку нагрузки между модулями.

Плоскость управления занимается одной частью работы всей системы, поэтому, чтобы полностью понимать, как все происходит в кластере Kubernetes, необходимо также понимать, что делает агент Kubelet и служебный прокси Kubernetes. Мы узнаем об этом дальше.

11.1.7 Что делает агент Kubelet

В отличие от всех контроллеров, которые являются частью плоскости управления Kubernetes и выполняются на ведущем узле (узлах), агент Kubelet и служебный прокси выполняются на рабочих узлах, где работают фактические контейнеры модулей. Что именно делает агент Kubelet?

Задача агента Kubelet

Если говорить коротко, агент Kubelet – это компонент, отвечающий за все, что выполняется на рабочем узле. Его первоначальная задача – зарегистрировать узел, на котором он работает, путем создания ресурса узла на сервере API. Затем он должен непрерывно отслеживать сервер API для модулей, которые были назначены на этот узел, и запускать контейнеры модуля. Он это делает, поручая сконфигурированной среде выполнения контейнеров (то есть платформе Docker, CoreOS платформы rkt или чему-то еще) запустить контейнер из конкретного образа контейнера. Затем агент Kubelet постоянно отслеживает запущенные контейнеры и сообщает об их статусе, событиях и потреблении ресурсов серверу API.

Агент Kubelet также является тем компонентом, который выполняет проверки живучести контейнеров, перезапуская контейнеры, когда проверки не срабатывают. Наконец, он завершает работу контейнеров, когда их модуль удаляется из сервера API, и уведомляет сервер о том, что модуль прекратил работу.

Запуск статических модулей без сервера API

Несмотря на то что агент Kubelet обменивается с сервером API Kubernetes и получает оттуда манифесты модуля, он также может запускать модули на основе файлов манифеста модуля в конкретном локальном каталоге, как показано на рис. 11.8. Этот функционал используется для запуска контейнеризированных версий компонентов плоскости управления в виде модулей, как вы убедились в начале этой главы.

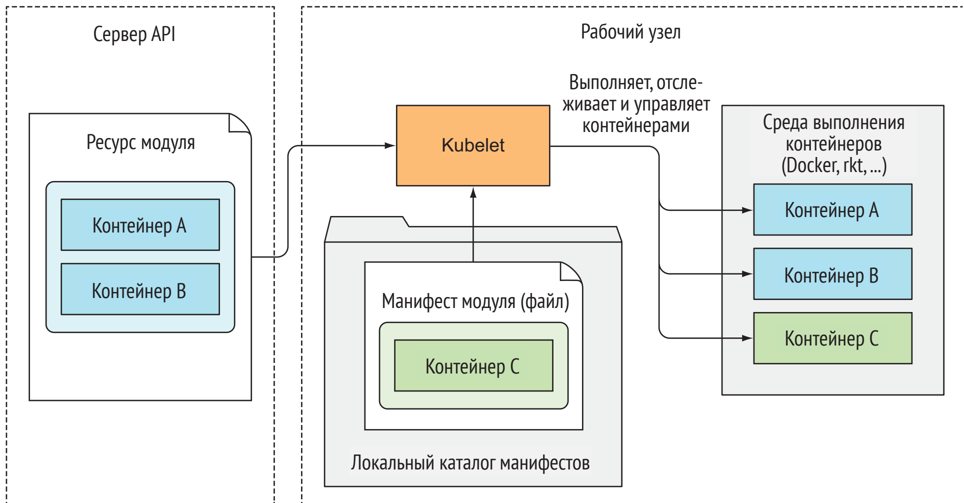


Рис. 11.8. Агент Kubelet запускает модули на основе спецификаций модулей с сервера API и локального файлового каталога

Вместо того чтобы изначально запускать системные компоненты Kubernetes, вы можете поместить их манифесты модуля в каталог манифестов аген-

та Kubelet и давать агенту Kubelet запускать их и управлять ими. Вы также можете использовать тот же метод для запуска ваших собственных системных контейнеров. Однако это рекомендуется делать посредством набора демонов DaemonSet.

11.1.8 Роль служебного сетевого прокси системы Kubernetes

Кроме агента Kubelet, каждый рабочий узел также выполняет сетевой прокси kube-proxy, цель которого – убедиться, что клиенты могут подключаться к службам, которые вы определяете посредством API Kubernetes. Сетевой прокси kube-proxy гарантирует, что подключения к IP-адресу и порту службы в итоге окажутся в одном из модулей, привязанных к службе (или других, немодульных, конечных точках службы). Когда служба поддерживается несколькими модулями, прокси выполняет балансировку нагрузки между этими модулями.

Почему он называется прокси

Первоначальная реализация kube-proxy была прокси-сервером в пользовательском пространстве ОС. Это был процесс сервера, который принимал подключения и проксировал их в модули. Для перехвата подключений, предназначенных для IP-адресов службы, прокси конфигурировал правила `iptables` (IP-таблицы `iptables` – это инструмент для управления функциями фильтрации пакетов ядра Linux) для перенаправления подключений на прокси-сервер. Примерная схема режима прокси `userspace` показана на рис. 11.9.

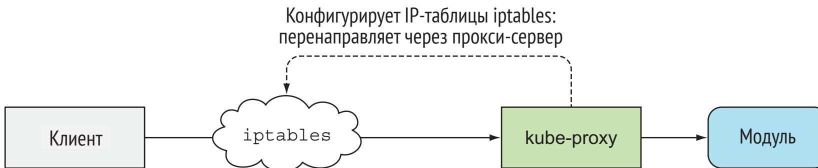


Рис. 11.9. Режим прокси userspace

Kube-proxy получил свое имя, потому что он представлял собой фактический прокси. Однако текущая, гораздо более эффективная реализация для перенаправления пакетов на случайно выбранный внутренний модуль без передачи их через фактический прокси-сервер использует только правила `iptables`. Этот режим называется режимом прокси `iptables` и показан на рис. 11.10.

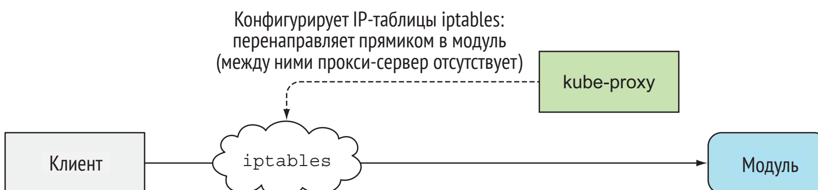


Рис. 11.10. Режим прокси iptables

Основное различие между этими двумя режимами заключается в том, проходят ли пакеты через kube-proxy и должны ли они обрабатываться в пользовательском пространстве, или же они обрабатываются только ядром (в пространстве ядра). Это существенно влияет на производительность.

Еще одно меньшее различие заключается в том, что режим прокси `userspace` балансировал подключения между модулями в истинном циклическом режиме, в то время как режим прокси `iptables` этого не делает – он выбирает модули случайным образом. Когда служба используется всего несколькими клиентами, они не могут быть распределены равномерно по модулям. Например, если у службы есть два поддерживающих модуля, но только пять или около того клиентов, то не удивляйтесь, если вы увидите, что четыре клиента подключаются к модулю A и только один клиент подключается к модулю B. В случае большего количества клиентов или модулей эта проблема не так очевидна.

Вы узнаете, как работает режим прокси `iptables`, в разделе 11.5.

11.1.9 Знакомство с надстройками Kubernetes

Мы обсудили ключевые компоненты, которые выполняют работу кластера Kubernetes. Но в начале главы мы также перечислили несколько надстроек, которые, хотя и не всегда требуются, включают такие функциональные средства, как DNS-поиск служб Kubernetes, предоставление нескольких служб HTTP через один внешний IP-адрес, веб-панель мониторинга Kubernetes и т. д.

Как развертываются надстройки

Эти компоненты доступны в виде надстроек и развертываются в виде модулей путем отправки манифестов YAML на сервер API, как вы делали в течение всей этой книги. Некоторые из этих компонентов развертываются через ресурс развертывания Deployment или ресурс контроллера репликации ReplicationController, а некоторые – через набор демонов DaemonSet.

Например, во время написания книги, в Minikube, надстройки контроллера входа Ingress и панели мониторинга развертываются как контроллеры репликации, как показано в следующем ниже листинге.

Листинг 11.7. Надстройки, развернутые с помощью контроллеров репликации в Minikube

```
$ kubectl get rc -n kube-system
NAME                       DESIRED  CURRENT  READY  AGE
default-http-backend       1        1        1      6d
kubernetes-dashboard       1        1        1      6d
nginx-ingress-controller   1        1        1      6d
```

Надстройка DNS развертывается как развертывание Deployment, как показано в следующем ниже листинге.

Листинг 11.8. Развертывание kube-dns

```
$ kubectl get deploy -n kube-system
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
kube-dns  1        1        1           1          6d
```

Давайте посмотрим, как работают DNS и контроллеры Ingress.

Как работает DNS-сервер

Все модули в кластере по умолчанию настроены на использование внутреннего DNS-сервера кластера. Это позволяет модулям легко отыскивать службы по имени или даже по IP-адресам модуля в случае безголовых служб.

Модуль DNS-сервера предоставляется через службу `kube-dns`, что позволяет перемещать модуль по кластеру, как и любой другой модуль. IP-адрес службы указан в качестве `nameserver` в файле /etc/resolv.conf внутри каждого контейнера, развернутого в кластере. Модуль `kube-dns` использует механизм отслеживания сервера API для наблюдения за изменениями служб и конечных точек и обновляет свои ресурсные записи DNS с каждым изменением, позволяя своим клиентам всегда получать (относительно) актуальную информацию DNS. Здесь слово «относительно» использовано потому, что во время между обновлением ресурса службы или конечных точек и временем, когда модуль DNS получает уведомление от наблюдения, записи DNS могут быть неактуальными.

Как работают контроллеры Ingress (большинство из них)

В отличие от надстройки DNS, вы найдете несколько различных реализаций контроллеров входа Ingress, но большинство из них работает одинаково. Контроллер входа запускает обратный прокси-сервер (например, Nginx) и держит его сконфигурированным в соответствии с ресурсами входа Ingress, службы Service и конечных точек Endpoints, определенными в кластере. Этот контроллер, следовательно, должен наблюдать за данными ресурсами (опять же, через механизм наблюдения) и изменять конфигурацию прокси-сервера каждый раз, когда один из них изменяется.

Хотя в определении ресурса Ingress указывается на службу, контроллеры Ingress перенаправляют трафик в модуль службы напрямую, а не через IP-адрес службы. Это влияет на сохранность клиентских IP-адресов, когда внешние клиенты подключаются через контроллер Ingress, что в определенных случаях использования делает их предпочтительными, по сравнению со службами.

Использование других надстроек

Вы увидели, что и DNS-сервер, и надстройки контроллера Ingress похожи на контроллеры, работающие в менеджере контроллеров, за одним исключением – вместо того чтобы только наблюдать и изменять ресурсы через сервер API, они также принимают клиентские подключения.

Другие надстройки похожи. Все они должны наблюдать за состоянием кластера и выполнять необходимые действия при его изменении. В этой и остальных главах мы представим несколько других надстроек.

11.1.10 Все воедино

Вы теперь узнали, что вся система Kubernetes состоит из относительно небольших, слабо сопряженных компонентов с хорошим разделением компетенции. Сервер API, планировщик, отдельные контроллеры, работающие внутри менеджера контроллеров, агент Kubelet и kube-proxy работают вместе, чтобы удерживать фактическое состояние системы в синхронизации с тем, что вы задаете в качестве желаемого состояния.

Например, отправка манифеста модуля на сервер API инициирует скоординированный танец различных компонентов Kubernetes, который в конечном итоге приводит к запуску контейнеров модуля. Вы узнаете, как этот танец разворачивается, в следующем разделе.

11.2 Взаимодействие контроллеров

Сейчас вы знаете обо всех компонентах, из которых состоит кластер Kubernetes. Теперь, чтобы укрепить ваше понимание того, как работает Kubernetes, давайте рассмотрим, что происходит, когда создается ресурс модуля Pod. Поскольку вы, как правило, не создаете модули напрямую, вы создадите ресурс развертывания и увидите все, что должно произойти для запуска контейнеров модуля.

11.2.1 Какие компоненты задействованы

Еще до запуска всего процесса контроллеры, планировщик и агент Kubelet отслеживают изменения в соответствующих типах ресурсов на сервере API. Это показано на рис. 11.11. Изображенные на рисунке компоненты будут играть определенную роль в процессе, который вы собираетесь запустить. Данная схема не включает хранилище etcd, потому что оно скрыто за сервером API, и поэтому вы можете рассматривать сервер API как место, где хранятся объекты.

11.2.2 Цепь событий

Представьте, что вы подготовили файл YAML, содержащий манифест развертывания, и собираетесь отправить его в Kubernetes через `kubectl`. Инструмент `kubectl` отправляет манифест серверу API Kubernetes в запросе HTTP POST. Сервер API проверяет спецификацию развертывания, сохраняет ее в хранилище etcd и возвращает ответ инструменту `kubectl`. Теперь цепь событий начинает разворачиваться, как показано на рис. 11.12.

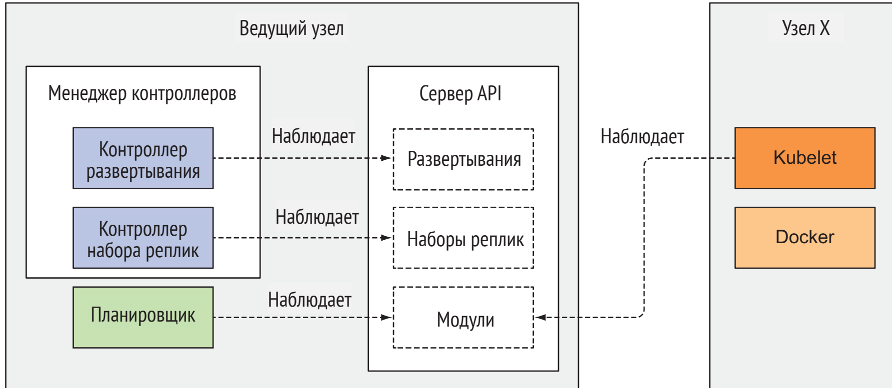


Рис. 11.11. Компоненты Kubernetes, наблюдающие за объектами API с помощью API сервера

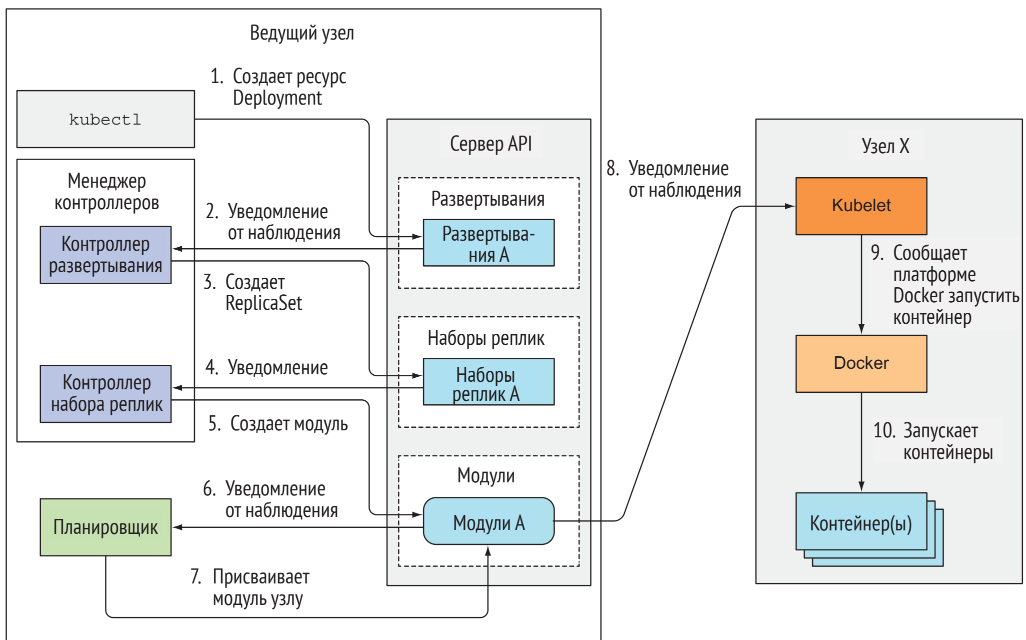


Рис. 11.12. Цепь событий, которая разворачивается, когда ресурс развертывания размещается в сервере API

Контроллер развертывания создает набор реплик

Все клиенты сервера API, просматривающие список развертываний через механизм наблюдения сервера API, получают уведомление о вновь созданном ресурсе развертывания сразу после его создания. Одним из таких клиентов является контроллер развертывания, который, как мы обсуждали ранее, является активным компонентом, ответственным за обработку развертываний.

Как вы помните из главы 9, развертывание работает с помощью одного или нескольких наборов реплик ReplicaSet, которые затем создают фактиче-

ские модули. Когда контроллер развертывания обнаруживает новый объект развертывания Deployment, он создает набор реплик ReplicaSet для текущей спецификации развертывания. Это предполагает создание нового ресурса ReplicaSet через API Kubernetes. Контроллер развертывания вообще не работает с отдельными модулями.

Контроллер набора реплик создает ресурсы модулей

Вновь созданный набор реплик ReplicaSet затем подхватывается контроллером ReplicaSet, который следит за созданием, изменением и удалением ресурсов ReplicaSet на сервере API. Данный контроллер принимает во внимание количество реплик и селектор модулей, определенный в наборе ReplicaSet, и проверяет, достаточно ли существующих модулей соответствует селектору.

Затем контроллер создает ресурсы модулей на основе шаблона модуля в наборе ReplicaSet (шаблон модуля был скопирован из объекта развертывания Deployment, когда контроллер развертывания создал набор ReplicaSet).

Планировщик назначает узел вновь созданным модулям

Эти недавно созданные модули теперь хранятся в хранилище etcd, но у каждого из них до сих пор нет одной важной вещи – с ними еще не связан узел. Их атрибут `nodeName` не задан. Планировщик наблюдает за такими модулями и, когда он встречает один такой модуль, выбирает для модуля самый лучший узел и назначает модуль узлу. Определение модуля теперь включает имя узла, на котором он должен работать.

До сих пор все происходило в плоскости управления Kubernetes. Ни один из контроллеров, принимавших участие во всем этом процессе, не сделал ничего осязаемого, кроме обновления ресурсов через сервер API.

Kubelet запускает контейнеры модуля

До этого момента рабочие узлы ничего не делали. Контейнеры модуля еще не запущены. Образы для контейнеров модуля еще даже не скачаны.

Но теперь, когда модуль назначен определенному узелу, агент Kubelet на этом узле может, наконец, приступить к работе. Агент Kubelet, наблюдая за изменениями в модулях на сервере API, видит, что новый модуль назначен его узлу, и поэтому он верифицирует определение модуля и поручает платформе Docker, или любой другой среде выполнения контейнеров, которую он использует, запустить контейнеры модуля. Затем среда выполнения контейнеров запускает контейнеры.

11.2.3 Наблюдение за событиями кластера

При выполнении этих действий компоненты плоскости управления и агент Kubelet передают события серверу API. Они делают это путем создания ресурсов Event, которые аналогичны любому другому ресурсу Kubernetes. Вы уже видели события, относящиеся к определенным ресурсам, когда вы применяли команду `kubectl describe` для инспектирования этих ресурсов, но вы также

можете получать события непосредственно с помощью команды `kubectl get events`.

Возможно, это касается только меня, но использовать команду `kubectl get` для инспектирования событий – довольно болезненная работа, потому что события не отображаются в правильном временном порядке. Вместо этого, если событие происходит несколько раз, это событие отображается всего один раз, показывая, когда оно было замечено впервые, когда оно было замечено в последний раз и количество раз, когда это произошло. К счастью, наблюдать за событиями с помощью параметра `--watch` намного проще для глаз и полезно для просмотра того, что происходит в кластере.

В следующем ниже листинге показаны события, созданные в описанном ранее процессе (некоторые столбцы были удалены, а результаты сильно изменены, чтобы сделать их удобочитаемыми в ограниченном пространстве страницы книги).

Листинг 11.9. Просмотр событий, генерируемых контроллерами

```
$ kubectl get events --watch
    NAME             KIND        REASON              SOURCE
... kubia            Deployment  ScalingReplicaSet   deployment-controller
                     ➥ Scaled up replica set kubia-193 to 3
... kubia-193        ReplicaSet  SuccessfulCreate    replicaset-controller
                     ➥ Created pod: kubia-193-w7ll2
... kubia-193-tpg6j  Pod         Scheduled           default-scheduler
                     ➥ Successfully assigned kubia-193-tpg6j to node1
... kubia-193        ReplicaSet  SuccessfulCreate    replicaset-controller
                     ➥ Created pod: kubia-193-39590
... kubia-193        ReplicaSet  SuccessfulCreate    replicaset-controller
                     ➥ Created pod: kubia-193-tpg6j
... kubia-193-39590  Pod         Scheduled           default-scheduler
                     ➥ Successfully assigned kubia-193-39590 to node2
... kubia-193-w7ll2  Pod         Scheduled           default-scheduler
                     ➥ Successfully assigned kubia-193-w7ll2 to node2
... kubia-193-tpg6j  Pod         Pulled              kubelet, node1
                     ➥ Container image already present on machine
... kubia-193-tpg6j  Pod         Created             kubelet, node1
                     ➥ Created container with id 13da752
... kubia-193-39590  Pod         Pulled              kubelet, node2
                     ➥ Container image already present on machine
... kubia-193-tpg6j  Pod         Started             kubelet, node1
                     ➥ Started container with id 13da752
... kubia-193-w7ll2  Pod         Pulled              kubelet, node2
                     ➥ Container image already present on machine
... kubia-193-39590  Pod         Created             kubelet, node2
                     ➥ Created container with id 8850184
```

Как вы можете видеть, столбец SOURCE показывает контроллер, выполняющий действие, и столбцы NAME и KIND показывают ресурс, на который действует контроллер. Столбец причины REASON и столбец сообщения MESSAGE (показанный в каждой второй строке) предоставляют более подробную информацию о том, что сделал контроллер.

11.3 Что такое запущенный модуль

Теперь, когда модуль запущен, давайте рассмотрим, что из себя представляет запущенный модуль. Если модуль содержит одиночный контейнер, как вы думаете, агент Kubelet запускает только этот одиночный контейнер или же есть что-то еще?

На протяжении всей книги вы запускали несколько модулей. Если вы – из тех людей, которые любят добираться до сути, вы, возможно, уже подсмотрели, что в точности Docker выполнял, когда вы создавали модуль. Если нет, давайте объясню, чтобы вы поняли.

Представьте, что вы запускаете один контейнер. Предположим, вы создаете модуль Nginx:

```
$ kubectl run nginx --image=nginx
deployment "nginx" created
```

Теперь можно войти в рабочий узел по ssh, на котором выполняется модуль, и проверить список запущенных контейнеров Docker. Для того чтобы это протестировать, я использую Minikube, поэтому, чтобы войти по ssh в одиночный узел, я использую команду minikube ssh. Если вы используете GKE, то можете войти по ssh в узел с помощью команды gcloud compute ssh <имя узла>.

После того как вы оказались внутри узла, вы можете вывести список всех запущенных контейнеров с помощью команды docker ps, как показано в следующем ниже листинге.

Листинг 11.10. Список запущенных контейнеров Docker

```
docker@minikubeVM:~$ docker ps

CONTAINER ID  IMAGE                COMMAND                CREATED
c917a6f3c3f7  nginx                "nginx -g 'daemon off"  4 seconds ago
98b8bf797174  gcr.io/.../pause:3.0  "/pause"               7 seconds ago
```

ПРИМЕЧАНИЕ. Из приведенного выше листинга я убрал ненужную информацию – она включает в себя и столбцы, и строки. Я также удалил все остальные запущенные контейнеры. Если вы тестируете это самостоятельно, то обратите внимание на два контейнера, которые были созданы несколько секунд назад.

Как и ожидалось, вы видите контейнер Nginx, а также дополнительный контейнер. Судя по столбцу COMMAND, этот дополнительный контейнер ничего не

делает (команда контейнера равна `"pause"`). Если присмотреться, то вы увидите, что этот контейнер был создан за несколько секунд до контейнера Nginx. Какова же его роль?

Этот приостановочный контейнер `pause` является контейнером, который собирает все контейнеры модуля вместе. Напомним, что все контейнеры модуля делят между собой одну и ту же сеть и другие пространства имен Linux. Контейнер `pause` – это инфраструктурный контейнер, единственной целью которого является хранение всех этих пространств имен. Все другие определяемые пользователем контейнеры модуля далее используют пространства имен инфраструктурного контейнера модуля (см. рис. 11.13).

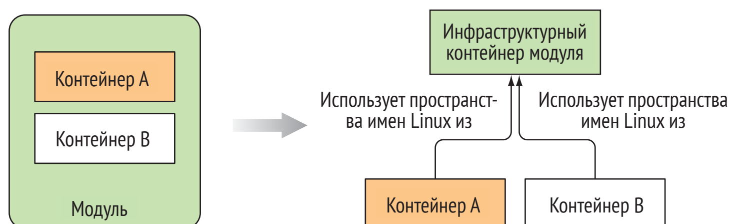


Рис. 11.13. Двухконтейнерный модуль приводит к трем работающим контейнерам, делящим между собой одни и те же пространства имен Linux

Фактические прикладные контейнеры могут умереть и быть перезапущены. Когда такой контейнер запускается снова, он должен стать частью тех же пространств имен Linux, что и раньше. Инфраструктурный контейнер делает это возможным, так как его жизненный цикл привязан к жизненному циклу модуля – этот контейнер выполняется с момента назначения модуля узлу до удаления модуля. Если по ходу инфраструктурный модуль будет убит, то агент Kubelet воссоздаст его и все контейнеры модуля.

11.4 Интермодульное сетевое взаимодействие

Теперь вы знаете, что каждый модуль получает свой собственный уникальный IP-адрес и может взаимодействовать со всеми другими модулями через плоскую сеть без NAT. Как именно Kubernetes этого достигает? Если коротко, то он этого не делает. Сеть настраивается не самим Kubernetes, а системным администратором или плагином контейнерного сетевого интерфейса Container Network Interface (CNI).

11.4.1 Как должна выглядеть сеть

Система Kubernetes не требует, чтобы вы использовали определенную сетевую технологию, но она требует, чтобы модули (или, если точнее, их контейнеры) могли взаимодействовать друг с другом независимо от того, работают они на одном рабочем узле или нет. Сеть, которую модули используют для взаимодействия, должна быть такой, чтобы IP-адрес, который модуль видит как его

собственный, был в точности тем же адресом, который видят все остальные модули как IP-адрес рассматриваемого модуля.

Посмотрите на рис. 11.14. Когда модуль A подключается к модулю B (то есть передает ему сетевой пакет), исходный IP-адрес, который видит модуль B, должен быть тем же IP-адресом, который модуль A видит как свой собственный. Между ними не должно выполняться никакого преобразования сетевых адресов (NAT) – пакет, передаваемый модулем A, должен достигнуть модуля B, и при этом и исходный, и целевой адреса должны оставаться без изменений.

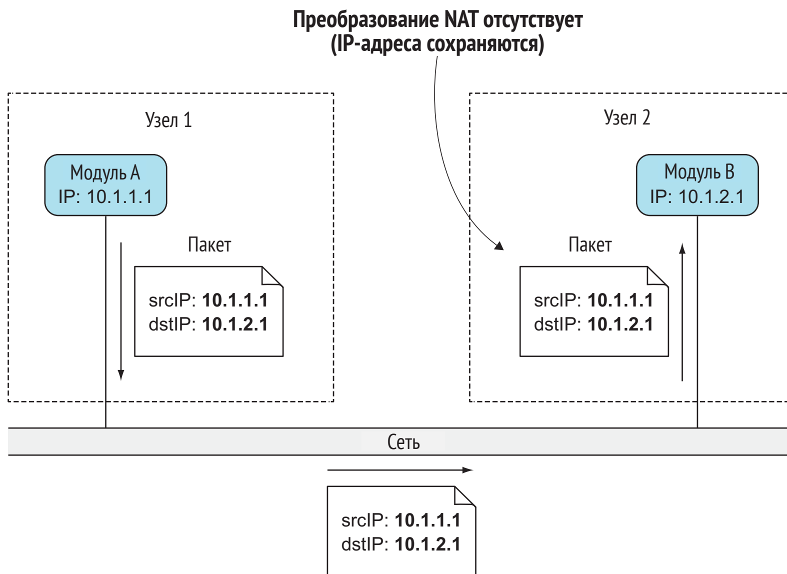


Рис. 11.14. Kubernetes предписывает, чтобы модули подключались друг к другу через сеть без преобразования NAT

Это важно, потому что упрощает сетевое взаимодействие для приложений, работающих внутри модулей, и в точности соответствует тому, как если бы они работали на машинах, подключенных к тому же самому сетевому коммутатору. Отсутствие преобразования NAT между модулями позволяет приложениям, работающим внутри них, саморегистрироваться в других модулях.

Например, предположим, что у вас есть клиентский модуль X и модуль Y, который предоставляет своего рода службу уведомлений для всех модулей, которые в нем регистрируются. Модуль X подключается к модулю Y и сообщает ему: «Эй, я – модуль X, доступный по IP 1.2.3.4; пожалуйста, отправьте мне обновления по этому IP-адресу». Предоставляющий службу модуль может подключиться к первому модулю с помощью полученного IP-адреса.

Требование к обмену между модулями без преобразования NAT распространяется также на взаимодействие модуль–узел и узел–модуль. Но когда модуль обменивается со службами по интернету, исходный IP-адрес пакетов, которые отправляет модуль, не требуется изменять, потому что IP-адрес мо-

дуля является приватным. Источниковый IP-адрес исходящих пакетов заменяется на IP-адрес хоста рабочего узла.

Создание надлежащего кластера Kubernetes включает в себя настройку сети в соответствии с этими требованиями. Для этого существуют различные методы и технологии, каждый со своими преимуществами или недостатками в заданном сценарии. По этой причине мы не будем вдаваться в конкретные технологии. Вместо этого давайте лучше поясним, как работает межмодульное сетевое взаимодействие в целом.

11.4.2 Более детальное рассмотрение работы сетевого взаимодействия

В разделе 11.3 мы увидели, что IP-адрес модуля и пространство сетевых имен настраиваются и удерживаются инфраструктурным контейнером (контейнером `pause`). И далее контейнеры модуля используют его пространство сетевых имен. Следовательно, сетевой интерфейс модуля – это все, что настроено в инфраструктурном контейнере. Давайте посмотрим, как создается этот интерфейс и как он подключается к интерфейсам во всех других модулях. Взгляните на рис. 11.15. Мы обсудим это дальше.

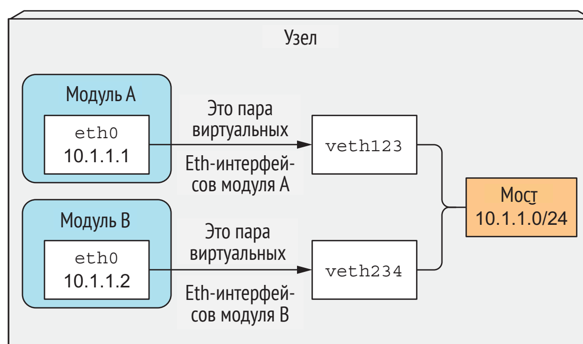


Рис. 11.15. Модули на узле связаны с одним и тем же мостом через пары виртуальных Ethernet-интерфейсов

Включение коммуникации между модулями на одном узле

Перед запуском инфраструктурного контейнера для контейнера создается пара виртуальных Ethernet-интерфейсов (пара veth). Один интерфейс пары остается в пространстве имен хоста (вы увидите его в списке как `vethXXX` при выполнении команды `ifconfig` на узле), тогда как другой перемещается в пространство сетевых имен контейнера и переименовывается в `eth0`. Два виртуальных интерфейса подобны двум концам трубы (или подобны двум сетевым устройствам, соединенным кабелем Ethernet) – то, что входит с одной стороны, выходит с другой, и наоборот.

Интерфейс в пространстве сетевых имен хоста присоединен к сетевому мосту, который сконфигурирован для использования средой выполнения

контейнера. Интерфейсу `eth0` в контейнере назначается IP-адрес из диапазона адресов моста. Все, что приложение, работающее внутри контейнера, отправляет в сетевой интерфейс `eth0` (который находится в пространстве имен контейнера), выходит на другом интерфейсе veth в пространстве имен узла и отправляется в мост. Это означает, что он может быть получен любым сетевым интерфейсом, подключенным к мосту.

Если модуль A отправляет сетевой пакет в модуль B, то пакет сначала проходит через пару veth модуля A в мост и затем через пару veth модуля B. Все контейнеры на узле подключены к одному мосту, имея в виду, что все они могут взаимодействовать друг с другом. Но для обеспечения взаимодействия между контейнерами, работающими на разных узлах, мосты на этих узлах должны быть каким-то образом подсоединены.

Включение коммуникации между модулями на разных узлах

Существует целый ряд способов подсоединения мостов на разных узлах. Это можно сделать с помощью оверлейной или андерлейной сети или же с помощью обычной маршрутизации уровня 3, которую мы рассмотрим далее.

Вы знаете, что IP-адреса модулей должны быть уникальными по всему кластеру, поэтому мосты по всем узлам должны использовать неперекрывающиеся диапазоны адресов, чтобы предотвращать получение модулями на других узлах одинакового IP-адреса. В примере, показанном на рис. 11.16, мост на узле A использует диапазон IP 10.1.1.0/24, и мост на узле B использует 10.1.2.0/24. Это гарантирует, что никаких конфликтов IP-адресов существовать не будет.

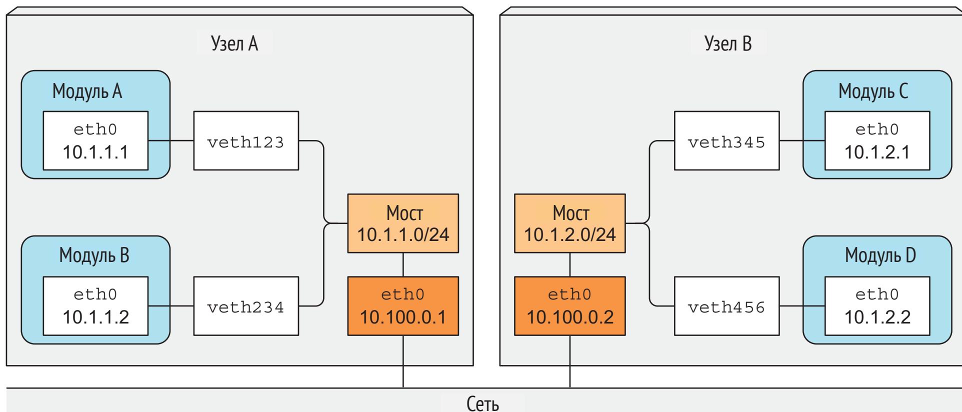


Рис. 11.16. Для того чтобы модули на других узлах взаимодействовали, мосты должны быть как-то связаны

Рисунок 11.16 показывает, что для включения коммуникации между модулями через два узла с простым сетевым взаимодействием уровня 3 физический сетевой интерфейс узла должен также быть связан с мостом. Табли-

цы маршрутизации на узле должны быть сконфигурированы так, чтобы все пакеты, предназначенные для 10.1.2.0/24, направлялись на узел B, тогда как таблицы маршрутизации узла B должны быть сконфигурированы так, чтобы пакеты, отправленные в 10.1.1.0/24, маршрутизировались в узел A.

При этом типе структуры, когда пакет отправляется контейнером на одном из узлов в контейнер на другом узле, пакет сначала проходит через пару veth, затем через мост к физическому адаптеру узла, потом по проводу к другому физическому адаптеру узла, через мост другого узла и, наконец, через пару veth целевого контейнера.

Это работает, только когда узлы подключены к одному сетевому коммутатору, без каких-либо маршрутизаторов между ними; иначе эти маршрутизаторы проигнорировали бы данные пакеты, потому что они ссылаются на приватные IP-адреса модулей. Разумеется, для маршрутизации пакетов между узлами можно сконфигурировать промежуточные маршрутизаторы, но это становится все более трудоемким и подверженным ошибкам, поскольку количество маршрутизаторов между узлами увеличивается. По этой причине проще использовать программно-определяемую сеть (SDN), которая делает узлы такими, как будто они подключены к одному и тому же сетевому коммутатору, независимо от фактической базовой топологии сети, какой бы сложной она ни была. Пакеты, отправляемые из модуля, инкапсулируются и отправляются по сети в узел, на котором работает другой модуль, где они деинкапсулируются и доставляются в модуль в их исходном виде.

11.4.3 Знакомство с контейнерным сетевым интерфейсом

Чтобы упростить подключение контейнеров к сети, был запущен проект под названием контейнерного сетевого интерфейса Container Network Interface (CNI). CNI позволяет конфигурировать систему Kubernetes для использования любого существующего плагина CNI. Эти плагины включают:

- Calico;
- Flannel;
- Romana;
- Weave Net;
- и др.

Мы не будем вдаваться в подробности этих плагинов; если вы хотите узнать о них больше, обратитесь к https://kubernetes.io/docs/concepts/cluster-administration/addons/.

Установка сетевого плагина не составляет труда. Вам нужно только развернуть YAML, содержащий набор DaemonSet и несколько других вспомогательных ресурсов. Этот YAML предоставляется на странице проекта каждого плагина. Как вы можете себе представить, набор DaemonSet используется для развертывания сетевого агента на всех узлах кластера. Затем он связывается с интерфейсом CNI на узле, но учтите, что для использования интерфейса CNI агент Kubelet должен быть запущен с параметром `--network-plugin=cni`.

11.5 Как реализованы службы

В главе 5 вы познакомились со службами, которые позволяют предоставлять набор модулей на долгоживущем, стабильном IP-адресе и порту. Для того чтобы сосредоточиться на том, для чего предназначены службы и как их можно использовать, мы преднамеренно не вдавались в то, как они работают. Но, чтобы по-настоящему разобраться в службах и лучше понять, где искать, когда вещи ведут себя не так, как вы ожидаете, вам нужно разобраться в том, как они реализованы.

11.5.1 Введение в kube-proxy

Всем, что связано со службами, занимается процесс kube-proxy, работающий на каждом узле. Изначально kube-proxy был фактическим прокси, ожидающим подключения и для каждого входящего подключения открывающим новое подключение с одним из модулей. Это было вызвано работой прокси в пользовательском режиме. Позже он был заменен на более эффективный режим прокси `iptables`. Теперь он используется по умолчанию, но если вы хотите, то можете сконфигурировать Kubernetes на использование старого режима.

Прежде чем мы продолжим, давайте кратко рассмотрим несколько аспектов относительно служб, которые имеют отношение к пониманию следующих моментов.

Мы узнали, что каждая служба имеет свой собственный стабильный IP-адрес и порт. Клиенты (обычно модули) используют службу, подключаясь к этому IP-адресу и порту. IP-адрес является виртуальным – он не приписан ни к каким сетевым интерфейсам и никогда не числится ни как исходный, ни как целевой IP-адрес в сетевом пакете, когда пакет покидает узел. Ключевая деталь служб состоит в том, что они состоят из пары IP-адреса и порта (или в случае многопортовых служб из нескольких пар IP-адресов и портов), поэтому IP-адрес службы сам по себе ничего не представляет. Вот почему их невозможно пинговать.

11.5.2 Как kube-proxy использует правила iptables

При создании службы на сервере API ей немедленно назначается виртуальный IP-адрес. Вскоре после этого сервер API уведомляет всех агентов kube-proxy, работающих на рабочих узлах, о создании новой службы. Затем каждый kube-proxy делает эту службу адресуемой на узле, на котором она запущена. Это делается путем настройки нескольких правил `iptables`, которые гарантируют, что каждый пакет, предназначенный для пары IP-адрес/порт службы, перехвачен и его целевой адрес модифицирован, с тем чтобы пакет перенаправлялся на один из модулей, поддерживающих службу.

Помимо просмотра изменений служб на сервере API, kube-proxy также следит за изменениями объектов конечных точек. Мы говорили о них в главе 5, но давайте освежим вашу память, поскольку легко забыть, что они даже су-

ществуют, потому что вы редко создаете их вручную. Объект конечных точек Endpoints содержит пары IP-адрес/порт всех модулей, которые поддерживают службу (пара IP-адрес/порт может также указывать на нечто другое, чем модуль). Поэтому kube-proxy также должен следить за всеми объектами конечных точек Endpoints. В конце концов, объект конечных точек изменяется всякий раз, когда создается или удаляется новый поддерживающий модуль и когда изменяется статус готовности модуля или изменяются метки модуля, и это попадает в область или выпадает из области действия службы.

Теперь давайте посмотрим на то, как kube-proxy позволяет клиентам подключаться к этим модулям через службу. Это показано на рис. 11.17.

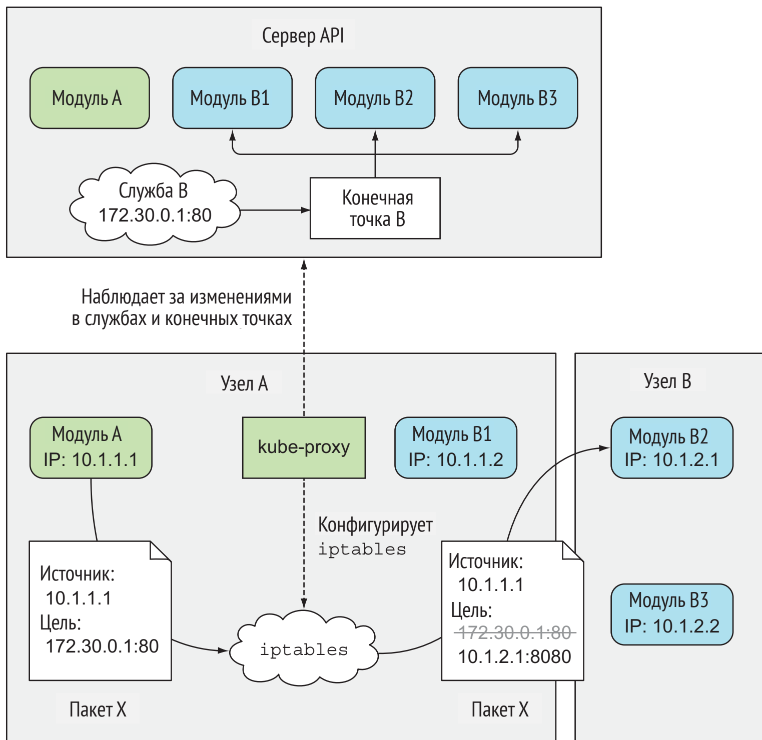


Рис. 11.17. Сетевые пакеты, отправляемые в виртуальную пару IP-адрес/порт службы, изменяются и перенаправляются в случайно выбранный внутренний модуль

На рисунке показано, что kube-proxy делает и как пакет, отправленный клиентским модулем, достигает одного из модулей, привязанных к службе. Давайте рассмотрим, что происходит с пакетом, когда он отправляется клиентским модулем (модулем A на рисунке).

Целевое назначение пакета первоначально установлено в IP-адрес и порт службы (в примере служба находится по 172.30.0.1:80). Перед отправкой в сеть пакет сначала обрабатывается ядром узла A в соответствии с правилами `iptables`, установленными на узле.

Ядро проверяет, соответствует ли пакет каким-либо из этих правил `iptables`. Одно из них говорит, что если какой-либо пакет имеет целевой IP-адрес, равный 172.30.0.1, и целевой порт, равный 80, то целевой IP-адрес пакета и порт должны быть заменены на IP-адрес и порт случайно выбранного модуля.

Пакет в примере совпадает с этим правилом, и поэтому целевой IP-адрес/порт изменяется. В примере модуль B2 был выбран случайным образом, поэтому целевой IP-адрес пакета меняется на 10.1.2.1 (IP-адрес модуля B2), а порт – на 8080 (целевой порт, указанный в спецификации службы). С данного момента происходит в точности так, как будто клиентский модуль отправил пакет в модуль B напрямую, а не через службу.

В действительности все немного сложнее, но это самое главное, что вы должны понимать.

11.6 Запуск высокодоступных кластеров

Одной из причин запуска приложений внутри Kubernetes является их бесперебойная работа без какого-либо или ограниченного ручного вмешательства в случае аварийных сбоев инфраструктуры. Для непрерывного выполнения служб все время должны быть работоспособны не только приложения, но и компоненты плоскости управления Kubernetes. Далее мы рассмотрим, с чем сопряжено достижение высокой доступности.

11.6.1 Обеспечение высокой доступности приложений

При запуске приложений в Kubernetes различные контроллеры обеспечивают, чтобы ваше приложение продолжало работать гладко и в указанном масштабе, даже если узлы аварийно прекращают свою работу. Чтобы обеспечить высокую доступность приложения, достаточно запустить его с помощью ресурса развертывания Deployment и настроить соответствующее количество реплик; обо всем остальном позаботится система Kubernetes.

Запуск нескольких экземпляров для снижения вероятности простоя

Для этого требуется, чтобы ваше приложение было горизонтально масштабируемым, но даже если ваше приложение таким не является, вам по-прежнему следует использовать развертывание с количеством реплик, равным единице. Если эта реплика становится недоступной, она будет быстро заменена новой, хотя это и не происходит мгновенно. На то, чтобы все задействованные контроллеры заметили аварийный сбой узла, создали новую реплику модуля и запустили контейнеры модуля, требуется время. Между ними неизбежно будет короткий период времени простоя.

Применение выборов лидера для немасштабируемых горизонтально приложений

Для того чтобы избежать простоя, необходимо запустить дополнительные неактивные реплики вместе с активными и использовать быстродействую-

щий механизм аренды или выбора лидера, который гарантирует, что активен только один. В случае если вы незнакомы с выборами лидеров, то это способ, позволяющий нескольким экземплярам приложений, работающих в распределенной среде, приходить к соглашению по вопросу, кто является лидером. Этот лидер является либо единственным, кто выполняет задачи, в то время как все остальные ждут, когда лидер аврийно прекратит работу, а затем сами станут лидерами, либо все они могут быть активными, при этом, к примеру, лидером является единственный экземпляр, который выполняет операции записи, в то время как все остальные предоставляют доступ к своим данным в режиме только для чтения. Это гарантирует, что два экземпляра никогда не выполняют одну и ту же работу, если это приводит к непредсказуемому поведению системы из-за условий состязательности.

Данный механизм не должен включаться непосредственно в само приложение. Вы можете использовать побочный контейнер, который выполняет все операции выбора лидера и сигнализирует главному контейнеру, когда он должен стать активным. Вы найдете пример выборов лидера в Kubernetes на https://github.com/kubernetes/contrib/tree/master/election.

Обеспечение высокой доступности ваших приложений достигается относительно просто, потому что Kubernetes берет на себя почти всю работу. Но что, если сама система Kubernetes потерпит аварийный сбой? Что, если серверы, выполняющие компоненты плоскости управления Kubernetes, упадут? Каким образом обеспечивается высокая доступность этих компонентов?

11.6.2 Обеспечение высокой доступности компонентов плоскости управления Kubernetes

В начале этой главы вы познакомились с несколькими компонентами, составляющими плоскость управления Kubernetes. Для того чтобы сделать Kubernetes высокодоступным, необходимо запустить несколько ведущих узлов, на которых выполняется несколько экземпляров следующих компонентов:

- etcd – распределенное хранилище данных, в котором хранятся все объекты API;
- сервер API;
- менеджер контроллеров – это процесс, в котором работают все контроллеры;
- планировщик.

Не вдаваясь в фактические сведения о том, как устанавливать и запускать эти компоненты, давайте посмотрим, что участвует в создании каждого из этих высокодоступных компонентов. На рис. 11.18 представлен обзор высокодоступного кластера.

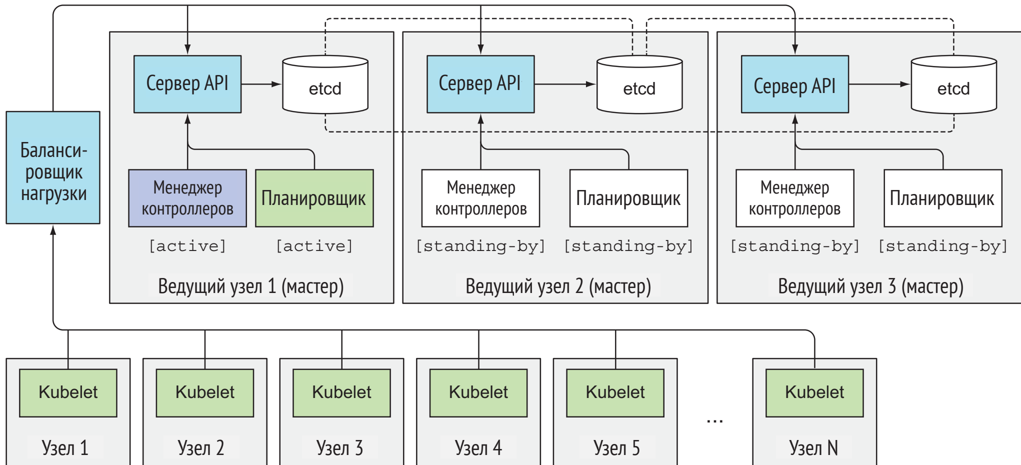


Рис. 11.18. Высокодоступный кластер с тремя ведущими узлами

Запуск кластера хранилища etcd

Поскольку хранилище etcd было спроектировано как распределенная система, одной из его ключевых особенностей является возможность запуска нескольких экземпляров etcd, поэтому сделать его высокодоступным – не такое уж большое дело. От вас лишь требуется запустить его на соответствующем количестве машин (три, пять или семь, как описано выше в этой главе) и сделать их осведомленными друг о друге. Для этого необходимо включить список всех остальных экземпляров в конфигурацию каждого экземпляра. Например, при запуске экземпляра указываются IP-адреса и порты, через которые можно получить доступ к другим экземплярам etcd.

Хранилище etcd реплицирует данные во всех своих экземплярах, поэтому сбой одного из узлов при запуске трехмашинного кластера по-прежнему позволит кластеру принимать операции чтения и записи. Для того чтобы повысить отказоустойчивость более чем до одного узла, вам нужно запустить пять или семь узлов etcd, что позволит кластеру обрабатывать соответственно два или три аварийных сбоя узлов. Наличие более семи экземпляров etcd почти никогда не требуется и начинает влиять на производительность.

Запуск множества экземпляров сервера API

Сделать сервер API высокодоступным еще проще. Поскольку сервер API (почти полностью) не имеет внутреннего состояния (все данные хранятся в etcd, при этом сервер API их все же кеширует), вы можете запускать столько серверов API, сколько вам нужно, и им вообще не нужно знать друг о друге. Обычно один сервер API размещается совместно с каждым экземпляром etcd. При этом экземплярам etcd не требуется какой-либо балансировщик нагрузки, поскольку каждый экземпляр сервера API обращается только к локальному экземпляру etcd.

С другой стороны, серверы API должны находиться под управлением балансировщика нагрузки, поэтому клиенты (`kubectl`, а также менеджер контроллеров, планировщик и все агенты Kubelet) всегда подключаются только к работоспособным экземплярам сервера API.

Обеспечение высокой доступности контроллеров и планировщика

По сравнению с сервером API, где несколько реплик могут работать одновременно, запустить несколько экземпляров менеджера контроллеров или планировщика не так-то просто. Поскольку все контроллеры и планировщик активно следят за состоянием кластера и действуют при его изменении, возможно, изменив состояние кластера дальше (например, во время увеличения количества требуемых реплик в наборе ReplicaSet на единицу контроллер набора ReplicaSet создает дополнительный модуль), выполнение нескольких экземпляров каждого из этих компонентов приведет к тому, что все они будут выполнять одно и то же действие. Они будут бежать наперегонки друг с другом, что может вызвать нежелательные эффекты (создание двух новых модулей вместо одного, как упоминалось в предыдущем примере).

По этой причине при запуске нескольких экземпляров этих компонентов в любой момент времени активен может быть только один экземпляр. К счастью, обо всем этом заботятся сами компоненты (это контролируется с помощью параметра `--leader-elect`, который по умолчанию имеет значение `true`). Каждый отдельный компонент будет активен только тогда, когда он будет выбран лидером. Только лидер выполняет фактическую работу, в то время как все другие экземпляры стоят и ждут, пока текущий лидер не прекратит работу аварийно. Когда это происходит, остальные экземпляры выбирают нового лидера, который затем берет на себя всю работу. Этот механизм обеспечивает, что два компонента никогда не работают в одно и то же время и не делают одинаковую работу (см. рис. 11.19).

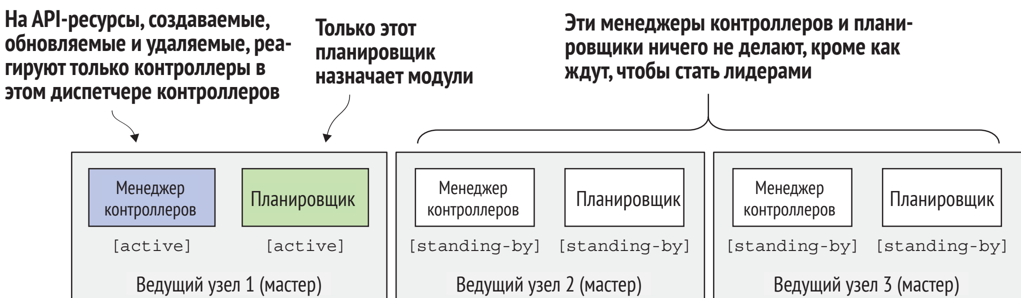


Рис. 11.19. Активны только один менеджер контроллеров и один планировщик; другие простаивают

Менеджер контроллеров и планировщик могут работать совместно с сервером API и хранилищем etcd, либо они могут выполняться на разных машинах. При совместном размещении они могут взаимодействовать с локальным сер-

вером API напрямую; в противном случае они подключаются к серверам API через балансировщик нагрузки.

Механизм выбора лидера, используемый компонентами плоскости управления

Наиболее интересным здесь я нахожу то, что эти компоненты, для того чтобы выбрать лидера, не должны обмениваться друг с другом напрямую. Механизм выборов лидера работает исключительно путем создания ресурса на сервере API. И это даже не особый вид ресурса – для достижения этого используется ресурс конечных точек (глагол «злоупотребляется», вероятно, является более подходящим).

Нет ничего особенного в том, что используется объект конечных точек Endpoints. Он применяется, потому что он не имеет побочных эффектов до тех пор, пока не существует службы с тем же именем. Может использоваться любой другой ресурс (в действительности механизм выборов лидера вскоре вместо конечных точек будет использовать словари конфигурации ConfigMap).

Уверен, что вас интересует то, как ресурс может быть применен для этой цели. Возьмем, например, планировщик. Все экземпляры планировщика пытаются создать (и позже обновить) ресурс конечных точек под названием `kube-scheduler`. Как показано в следующем ниже листинге, вы найдете его в пространстве имен `kube-system`.

Листинг 11.11. Ресурс конечных точек kube-scheduler, используемый для выбора лидера

```
$ kubectl get endpoints kube-scheduler -n kube-system -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":
      ➥ "minikube","leaseDurationSeconds":15,"acquireTime":
      ➥ "2017-05-27T18:54:53Z","renewTime":"2017-05-28T13:07:49Z",
      ➥ "leaderTransitions":0}'
  creationTimestamp: 2017-05-27T18:54:53Z
  name: kube-scheduler
  namespace: kube-system
  resourceVersion: "654059"
  selfLink: /api/v1/namespaces/kube-system/endpoints/kube-scheduler
  uid: f847bd14-430d-11e7-9720-080027f8fa4e
subsets: []
```

Аннотация `control-plane.alpha.kubernetes.io/leader` является важной частью. Как вы можете видеть, она содержит поле `holderIdentity`, которое имеет имя текущего лидера. Первый экземпляр, которому удается проставить

там свое имя, становится лидером. Для того чтобы сделать это, экземпляры соревнуются друг с другом, но победитель всегда только один.

Помните понятие оптимистического параллелизма, которое мы объясняли ранее? Оно гарантирует, что если несколько экземпляров попытаются записать свое имя в ресурс, то только один из них сделает это успешно. В зависимости от того, была ли запись своего имени в ресурс успешной или нет, каждый экземпляр знает, является он лидером или нет.

Как только он становится лидером, он должен периодически обновлять ресурс (по умолчанию каждые две секунды), чтобы все другие экземпляры знали, что он по-прежнему жив. Когда лидер аварийно прекращает свою работу, другие экземпляры видят, что ресурс не обновлялся некоторое время, и пытаются стать лидером, записывая свое имя в ресурс. Просто, не правда ли?

11.7 Резюме

Надеюсь, эта глава была интересной, и она улучшила ваши познания о внутренней работе Kubernetes. Эта глава показала:

- какие компоненты составляют кластер Kubernetes и за что отвечает каждый компонент;
- как сервер API, планировщик, различные контроллеры, работающие в менеджере контроллеров, и агент Kubelet работают вместе, чтобы оживлять модуль;
- как инфраструктурный контейнер связывает вместе все контейнеры модуля;
- как модули обмениваются с другими модулями, работающими на том же узле, через сетевой мост, и как эти мосты на разных узлах соединяются, для того чтобы модули, работающие на разных узлах, могли обмениваться между собой;
- как kube-proxy выполняет балансировку нагрузки между модулями в одной службе путем конфигурирования правил `iptables` на узле;
- как множества экземпляров каждого компонента плоскости управления могут выполняться, для того чтобы делать кластер высокодоступным.

Далее мы рассмотрим, как защищать сервер API и, следовательно, кластер в целом.

Глава 12

Защита сервера API Kubernetes

Эта глава посвящена:

- аутентификации;
- учетным записям ServiceAccount и тому, зачем они используются;
- плагину управления ролевым доступом (RBAC);
- использованию ролей Role и ролевых привязок RoleBinding;
- использованию кластерных ролей ClusterRole и привязок кластерных ролей ClusterRoleBinding;
- устанавливаемым по умолчанию ролям и привязкам.

В главе 8 вы познакомились с тем, как работающие в модулях приложения могут взаимодействовать с сервером API для получения или изменения состояния развернутых в кластере ресурсов. Для аутентификации на сервере API вы использовали токен учетной записи службы ServiceAccount, смонтированный в модуле. В этой главе вы познакомитесь с тем, что такое учетные записи службы ServiceAccount и как конфигурировать для них разрешения, а также разрешения для других субъектов, использующих кластер.

12.1 Аутентификация

В предыдущей главе мы отметили, что сервер API может быть сконфигурирован с помощью одного или нескольких плагинов аутентификации (и то же самое верно для плагинов авторизации). Когда сервер API получает запрос, этот запрос проходит через список плагинов аутентификации, и каждый из них может исследовать запрос и попытаться определить, кто его отправляет. Первый плагин, который может извлечь эту информацию из запроса, возвращает обратно в ядро сервера API имя пользователя, идентификатор пользователя и группы, к которым принадлежит клиент. Сервер API перестает вызывать оставшиеся плагины аутентификации и переходит к этапу авторизации.

Существует несколько плагинов аутентификации. Они получают идентичность клиента следующими способами:

- из сертификата клиента;
- из токена аутентификации, переданного в заголовке http;
- в результате обычной HTTP-аутентификации;
- другими.

Плагины аутентификации активируются с помощью параметров командной строки при запуске сервера API.

12.1.1 Пользователи и группы

Плагин аутентификации возвращает имя пользователя и группу (группы) аутентифицируемого пользователя. Kubernetes нигде не хранит эту информацию; он использует ее, чтобы верифицировать, авторизован пользователь для выполнения действия или нет.

Пользователи

Kubernetes различает два вида клиентов, подключающихся к серверу API:

- реальные люди (пользователи);
- модули (если конкретнее, работающие внутри них приложения).

Оба этих типа клиентов аутентифицируются с помощью вышеупомянутых плагинов аутентификации. Подразумевается, что пользователи должны управляться внешней системой, такой как система единого входа (Single Sign On, SSO), но модули используют механизм, называемый *учетными записями служб*, которые создаются и хранятся в кластере как ресурсы ServiceAccount. Для сравнения, ни один ресурс не представляет учетные записи пользователей, имея в виду, что вы не можете создавать, обновлять или удалять пользователей через сервер API.

Мы не будем вдаваться в подробности того, как управлять пользователями, но мы подробно рассмотрим учетные записи служб, потому что они необходимы для запуска модулей. Дополнительные сведения о конфигурировании кластера для аутентификации пользователей смотрите в руководстве администратора кластера Kubernetes по адресу `http://kubernetes.io/docs/admin`.

Группы

Как обычные пользователи, так и учетные записи ServiceAccount могут принадлежать к одной или нескольким группам. Мы отметили, что плагин аутентификации возвращает группы вместе с именем пользователя и идентификатором пользователя. Группы используются для предоставления разрешений сразу нескольким пользователям, а не отдельным пользователям.

Возвращаемые плагином группы являются не чем иным, как строковыми значениями, представляющими произвольные имена групп, однако встроенные группы имеют особое значение:

- группа `system:unauthenticated` используется для запросов, где ни один из плагинов аутентификации не мог аутентифицировать клиента;

- группа `system:authenticated` автоматически назначается пользователю, успешно прошедшему аутентификацию;
- группа `system:serviceaccounts` охватывает все учетные записи ServiceAccount в системе;
- группа `system:serviceaccounts:<пространство имен>` включает в себя все учетные записи ServiceAccount в определенном пространстве имен.

12.1.2 Знакомство с учетными записями службы

Давайте поближе рассмотрим учетные записи ServiceAccount. Вы уже знаете, что, перед тем как клиентам будет разрешено выполнять операции на сервере API, сервер требует от них, чтобы они аутентифицировались. И вы уже видели, как модули могут аутентифицироваться, отправляя содержимое файла `/var/run/secrets/kubernetes.io/serviceaccount/token`, который монтируется в файловую систему каждого контейнера посредством тома `secret`.

Но что именно представляет собой этот файл? Каждый модуль связан с учетной записью службы, которая представляет идентичность работающего в модуле приложения. Файл токена содержит токен аутентификации учетной записи ServiceAccount. Когда приложение использует этот токен для подключения к серверу API, плагин аутентификации проверяет подлинность учетной записи ServiceAccount и передает имя пользователя учетной записи ServiceAccount обратно в ядро сервера API. Имена пользователей учетной записи ServiceAccount форматируются следующим образом:

```
system:serviceaccount:<пространство имен>:<имя учетной записи службы>
```

Сервер API передает это имя пользователя сконфигурированным плагинам аутентификации, которые определяют, разрешено приложению выполнять действие с помощью учетной записи ServiceAccount или нет.

Учетные записи службы – это не что иное, как способ проверки подлинности работающего в модуле приложения на сервере API. Как уже упоминалось, приложения делают это, передавая в запросе токен учетной записи ServiceAccount.

Ресурс ServiceAccount

Учетные записи ServiceAccount – это ресурсы, такие же, как модули, секреты, словари конфигурации и т. д., которые ограничиваются отдельными пространствами имен. Устанавливаемая по умолчанию учетная запись ServiceAccount создается автоматически для каждого пространства имен (именно их и использовали ваши модули все время).

Список учетных записей ServiceAccount можно вывести точно так же, как вы делаете это с другими ресурсами:

```
$ kubectl get sa
NAME     SECRETS  AGE
default  1        1d
```

ПРИМЕЧАНИЕ. Аббревиатурой для `serviceaccount` являются символы `sa`.

Как вы можете видеть, текущее пространство имен содержит учетную запись по умолчанию ServiceAccount. При необходимости могут быть добавлены дополнительные учетные записи служб. Каждый модуль связан ровно с одной учетной записью службы, но несколько модулей могут использовать одну и ту же учетную запись. Как видно на рис. 12.1, модуль может использовать учетную запись ServiceAccount только из того же пространства имен.

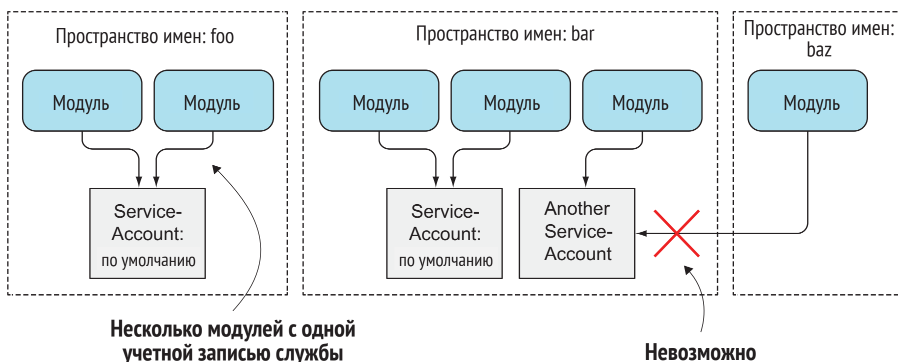


Рис. 12.1. Каждый модуль связан с одной учетной записью службы в пространстве имен модуля

Как учетные записи служб связаны с авторизацией

Вы можете назначить модулю учетную запись службы, указав имя учетной записи в манифесте модуля. Если вы не назначите ее явно, модуль будет использовать учетную запись службы по умолчанию в пространстве имен.

Назначая модулям различные учетные записи служб, вы можете контролировать то, к каким ресурсам имеет доступ каждый модуль. Когда запрос, несущий в себе токен аутентификации, поступает на сервер API, сервер использует этот токен для аутентификации клиента, отправляющего запрос, а затем определяет, разрешено ли соответствующей учетной записи службы выполнять запрошенную операцию. Сервер API получает эту информацию от общесистемного плагина авторизации, сконфигурированного администратором кластера. Одним из плагинов авторизации является плагин управления ролевым доступом (role-based access control, RBAC), который рассматривается далее в этой главе. Начиная с Kubernetes версии 1.6 плагин RBAC должен использоваться большинством кластеров.

12.1.3 Создание учетных записей ServiceAccount

Мы отметили, что для каждого пространства имен задана своя собственная, установленная по умолчанию учетная запись ServiceAccount, но при необходимости могут создаваться дополнительные. Но почему вы должны беспокоиться о создании учетных записей службы вместо использования тех, которые заданы для всех ваших модулей по умолчанию?

Очевидная причина – это безопасность кластера. Модули, которым не нужно читать метаданные кластера, должны работать в соответствии с ограниченной учетной записью, которая не позволяет им извлекать или изменять ресурсы, развернутые в кластере. Модули, которым нужно извлекать метаданные ресурса, должны работать в соответствии с учетной записью ServiceAccount, которая позволяет только читать метаданные этих объектов, в то время как модули, которым нужно эти объекты модифицировать, должны работать в соответствии со своей собственной учетной записью ServiceAccount, разрешающей производить модификацию объектов API.

Давайте посмотрим, каким образом создаются дополнительные учетные записи служб, как они соотносятся с секретами и как их можно назначать вашим модулям.

Создание учетной записи службы

Создать учетную запись службы невероятно легко благодаря специальной команде `kubectl create serviceaccount`. Давайте создадим новую учетную запись с именем `foo`:

```
$ kubectl create serviceaccount foo
serviceaccount "foo" created
```

Теперь вы можете проверить эту учетную запись с помощью команды `describe`, как показано в следующем ниже листинге.

Листинг 12.1. Проверка учетной записи службы с помощью команды kubectl describe

```
$ kubectl describe sa foo
Name:                foo
Namespace:           default
Labels:              <none>

Image pull secrets:  <none>

Mountable secrets:   foo-token-qzq7j

Tokens:              foo-token-qzq7j
```

С помощью этой учетной записи службы они будут добавлены автоматически ко всем модулям

Модули, использующие эту учетную запись службы, могут монтировать эти секреты только в том случае, если применяются монтируемые секреты

Токен(ы) аутентификации. Первый из них монтируется внутрь контейнера

Вы можете видеть, что секрет индивидуально настроенного токена был создан и связан с учетной записью службы. Если посмотреть на данные секрета с помощью команды `kubectl describe secret footoken-qzq7j`, то вы увидите, что он содержит те же элементы (сертификат CA, пространство имен и токен), что и у токена учетной записи службы по умолчанию (сам токен, очевидно, будет другим). Это показано в следующем ниже листинге.

Листинг 12.2. Инспектирование секрета индивидуально настроенной учетной записи службы

```
$ kubectl describe secret foo-token-qzq7j
...
ca.crt:         1066 bytes
namespace:      7 bytes
token:          eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

ПРИМЕЧАНИЕ. Вы, наверное, слышали о веб-токенах JSON (JWT). Токены аутентификации, используемые в учетных записях служб, являются маркерами JWT.

Монтируемые секреты учетной записи службы

Если проинспектировать учетную запись службы с помощью команды `kubectl describe`, то токен будет показан в списке монтируемых секретов (`Mountable secrets`). Следует объяснить, что этот список представляет. В главе 7 вы узнали, как создавать секреты и монтировать их внутри модуля. По умолчанию модуль может смонтировать любой секрет, который он захочет. Но учетная запись службы модуля может быть сконфигурирована так, чтобы разрешать модулю монтировать только те секреты, которые перечислены как монтируемые секреты в учетной записи службы. Чтобы активировать эту функциональную возможность, учетная запись службы ServiceAccount должна содержать следующую аннотацию: `kubernetes.io/enforce-mountable-secrets="true"`.

Если учетная запись ServiceAccount аннотируется этой аннотацией, любые использующие ее модули могут монтировать только монтируемые секреты учетных записей – они не могут использовать никакой другой секрет.

Секреты учетной записи для выгрузки образов

Учетная запись ServiceAccount также может содержать список секретов для выгрузки образов, которые мы рассмотрели в главе 7. Если вы не помните, то это секреты, которые содержат учетные данные для выгрузки образов контейнеров из приватного хранилища образов.

В следующем списке показан пример определения учетной записи, который включает секрет извлечения образа, созданный в главе 7.

Листинг 12.3. Учетная запись службы с секретом извлечения образа: sa-image-pull-secrets.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
imagePullSecrets:
- name: my-dockerhub-secret
```

Секреты учетной записи ServiceAccount для выгрузки образа ведут себя немного иначе, чем ее монтируемые секреты. В отличие от монтируемых секретов, они не определяют, какие секреты выгрузки образов используются модулем, а определяют, какие секреты автоматически добавляются во все модули с помощью учетной записи службы. Добавление секретов выгрузки образов в учетную запись ServiceAccount избавляет от необходимости добавлять их в каждый модуль по отдельности.

12.1.4 Назначение модулю учетной записи службы

После создания дополнительных учетных записей ServiceAccount вам нужно назначить их модулям. Это делается путем выставления в определении модуля имени учетной записи службы в поле `spec.serviceAccountName`.

> **ПРИМЕЧАНИЕ.** Учетная запись службы модуля должна быть установлена при создании модуля. Она не может быть изменена позже.

Создание модуля, использующего индивидуально настроенную учетную запись службы

В главе 8 вы развернули модуль, который выполнял контейнер на основе образа `tutum/curl` и контейнера-посредника рядом с ним. Вы использовали его для исследования интерфейса REST сервера API. Контейнер-посредник запустил процесс `kubectl proxy`, который использовал токен учетной записи модуля для аутентификации на сервере API.

Теперь можно этот модуль модифицировать, чтобы он использовал учетную запись `foo`, созданную несколько минут назад. В следующем ниже листинге показано определение данного модуля.

Листинг 12.4. Модуль, использующий учетную запись службы, действующую по умолчанию: curl-custom-sa.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: curl-custom-sa
spec:
  serviceAccountName: foo        <- Этот модуль вместо установленной по
  containers:                       умолчанию учетной записи использует
  - name: main                      учетную запись foo
    image: tutum/curl
    command: ["sleep", "9999999"]
  - name: ambassador
    image: luksa/kubectl-proxy:1.6.2
```

Для того чтобы получить подтверждение, что токен индивидуально настроенной учетной записи смонтирован в два контейнера, можно распечатать содержимое токена, как показано в следующем ниже листинге.

Листинг 12.5. Инспектирование токена, смонтированного в контейнер(ы) модуля

```
$ kubectl exec -it curl-custom-sa -c main
➥   cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

Вы можете увидеть, что этот токен принадлежит учетной записи `foo`, сравнив строку токена в листинге 12.5 со строкой в листинге 12.2.

Использование токена индивидуально настроенной учетной записи для обмена с сервером API

Давайте посмотрим, сможете ли вы обменяться информацией с сервером API, используя этот токен. Как упоминалось ранее, контейнер-посредник использует токен при обмене с сервером, поэтому вы можете проверить токен, пройдя через посла, который слушает на `localhost: 8001`. Это показано в следующем ниже листинге.

Листинг 12.6. Обмен с сервером API с помощью индивидуально настроенной учетной записи службы

```
$ kubectl exec -it curl-custom-sa -c main curl localhost:8001/api/v1/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
    "resourceVersion": "433895"
  },
  "items": [
  ...
```

Хорошо. Вы получили правильный отклик от сервера, что означает, что индивидуально настроенной учетной записи службы разрешено выводить список модулей. Это может быть вызвано тем, что ваш кластер не использует плагин авторизации RBAC или вы предоставили всем учетным записям полные разрешения, как было продемонстрировано в главе 8.

Когда ваш кластер не использует правильную авторизацию, создание и использование дополнительных учетных записей не имеет большого смысла, так как даже учетная запись, существующая по умолчанию, разрешает делать все, что угодно. В этом случае единственная причина использовать учетные записи состоит в том, чтобы усиливать монтируемые секреты или предоставлять секреты извлечения образов через учетную запись, как объяснено ранее.

Однако создание дополнительных учетных записей практически необходимо, когда вы используете плагин авторизации RBAC, который мы рассмотрим далее.

12.2 Защита кластера с помощью управления ролевым доступом

Начиная с версии Kubernetes 1.6.0 обеспечение безопасности кластера значительно возросло. В более ранних версиях, если вам удавалось получать токен аутентификации от одного из модулей, вы могли использовать его для того, чтобы делать в кластере все, что вы хотите. Если вы погуглите, то найдете демонстрации, показывающие, как атака *выхода за пределы пути* (или *обхода каталога*) (при которой клиенты могут получать файлы, расположенные за пределами корневого каталога веб-сервера) может быть использована для получения токена и его применения для запуска вредоносных модулей в небезопасном кластере Kubernetes.

Но в версии 1.8.0 плагин авторизации RBAC перешел в статус общей доступности (General Availability, GA) и теперь активирован по умолчанию на многих кластерах (например, при развертывании кластера с помощью `kubadm`, как описано в приложении B). Управление ролевым доступом RBAC предотвращает несанкционированный просмотр и изменение состояния кластеров. Учетная запись службы по умолчанию не может просматривать состояние кластера, не говоря уже об изменении его каким-либо образом, если только вы не предоставляете ей дополнительные привилегии. Для написания приложений, взаимодействующих с сервером API Kubernetes (как описано в главе 8), необходимо понимать, как управлять авторизацией с помощью RBAC-специфичных ресурсов.

ПРИМЕЧАНИЕ. Помимо RBAC, Kubernetes также содержит другие плагины авторизации, такие как плагин управления атрибутивно-ориентированным доступом (attribute-based access control, ABAC), плагин Web-Hook и собственные реализации плагинов. Правда, стандартным является плагин управления ролевым доступом RBAC.

12.2.1 Знакомство с плагином авторизации RBAC

Сервер API Kubernetes может быть сконфигурирован на использование плагина авторизации для проверки того, разрешено действие, которое пользователь запрашивает, или нет. Поскольку сервер API предоставляет доступ к интерфейсу REST, пользователи выполняют действия, отправляя на сервер запросы HTTP. Пользователи аутентифицируются, включая учетные данные в запрос (токен аутентификации, имя пользователя и пароль или сертификат клиента).

Действия

Какие же существуют действия? Как известно, клиенты REST отправляют запросы GET, POST, PUT, DELETE и другие типы HTTP-запросов по определенным URL-адресам, представляющим определенные ресурсы REST. В Kubernetes этими ресурсами являются модули, службы, секреты и т. д. Вот несколько примеров действий в Kubernetes:

- получить модули;
- создать службы;
- обновить секреты;
- и т. д.

Команды в этих примерах (get, create, update) увязываются с методами HTTP (GET, POST, PUT), выполняемыми клиентом (полная увязка показана в табл. 12.1). Существительные (модули, служба, секреты), разумеется, соответствуют ресурсам Kubernetes.

Плагин авторизации, такой как RBAC, который работает внутри сервера API, определяет, разрешено клиенту выполнять запрошенную команду на запрошенном ресурсе или нет.

Таблица 12.1. Сопоставление методов HTTP с командами авторизации

Метод HTTP	Глагол для одного ресурса	Глагол для коллекции
`GET`, `HEAD`	`get` (и `watch` для наблюдения)	`list` (и `watch`)
`POST`	`create`	отсутствует
`PUT`	`update`	отсутствует
`PATCH`	`patch`	отсутствует
`DELETE`	`delete`	`deletecollection`

> **ПРИМЕЧАНИЕ.** Дополнительный глагол use используется для ресурсов политики безопасности модулей PodSecurityPolicy, о которых рассказывается в следующей главе.

Помимо применения разрешений системы безопасности ко всем типам ресурсов, правила RBAC могут также применяться к определенным экземплярам ресурса (например, к службе `myservice`). Позже вы увидите, что разрешения также могут применяться к нересурсным URL-путям, поскольку не каждый путь, к которому сервер API предоставляет доступ, увязан с ресурсом (например, путь `/api` как таковой или информация о работоспособности сервера в `/healthz`).

Плагин авторизации RBAC

Плагин авторизации на основе управления ролевым доступом RBAC, как следует из названия, в качестве ключевого фактора в установлении того, мо-

жет пользователь выполнить действие или нет, использует роли пользователя. Субъект (который может быть человеком, учетной записью ServiceAccount или группой пользователей или учетных записей ServiceAccounts) связан с одной или несколькими ролями, и каждой роли разрешено выполнять определенные команды на определенных ресурсах.

Если пользователь имеет несколько ролей, то он может делать все, что ему позволяет любая из ролей. Если ни одна из ролей пользователя не содержит разрешения, например на обновление секретов, сервер API не позволит пользователю выполнять запросы PUT или PATCH на секретах.

Управлять авторизацией посредством плагина RBAC очень просто. Все делается путем создания четырех специфичных для RBAC ресурсов Kubernetes, которые мы рассмотрим далее.

12.2.2 Знакомство с ресурсами RBAC

Правила авторизации RBAC настраиваются с помощью четырех ресурсов, которые можно сгруппировать в две группы:

- роли Role и кластерные роли ClusterRole, которые задают, какие глаголы могут выполняться на ресурсах;
- привязки ролей RoleBinding и привязки кластерных ролей ClusterRoleBinding, которые привязывают вышеуказанные роли к определенным пользователям, группам или учетным записям ServiceAccount.

Роли определяют, что вообще можно делать, в то время как привязки определяют, кто может это делать (это показано на рис. 12.2).

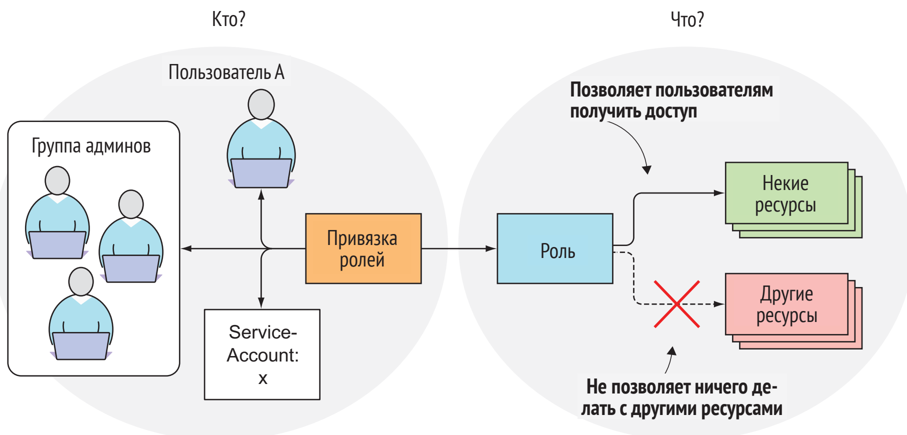


Рис. 12.2. Роли предоставляют разрешения, тогда как привязки ролей связывают роли с субъектами

Различие между ролью и кластерной ролью или между привязкой роли и привязкой кластерной роли состоит в том, что роль и привязка роли являются ресурсами, организованными в пространство имен, тогда как кластерная роль

и привязка кластерной роли являются ресурсами уровня кластера (не организованные в пространство имен). Это показано на рис. 12.3.

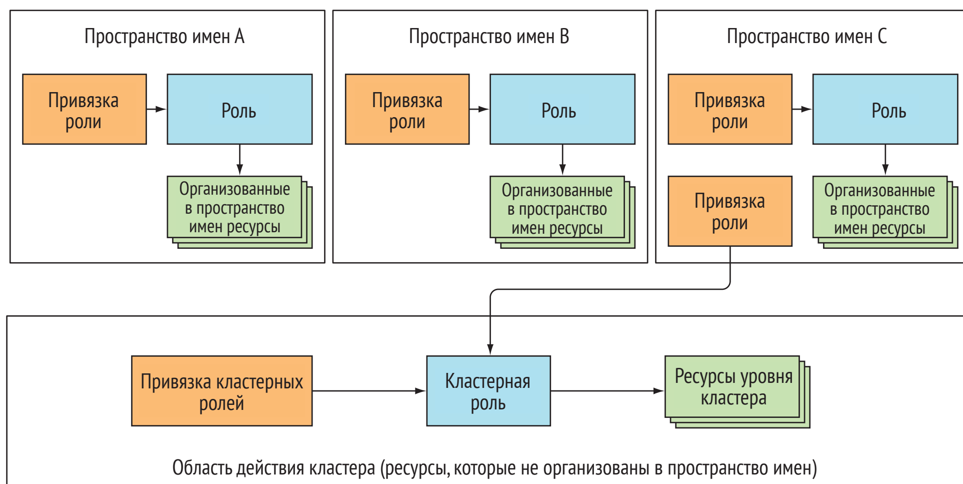


Рис. 12.3. Роли и привязки ролей организованы в пространство имен, кластерные роли и привязки кластерных ролей – нет

Как видно из рисунка, в одном пространстве имен может существовать несколько привязок ролей (это также верно для ролей). Аналогичным образом можно создать несколько привязок кластерных ролей и кластерных ролей. На рисунке показан еще один аспект, который состоит в том, что хотя привязки ролей организованы в пространство имен, они также могут ссылаться на кластерные роли, которые не имеют пространства имен.

Самый лучший способ узнать об этих четырех ресурсах и их последствиях – опробовать их в практическом упражнении. Этим вы сейчас и займетесь.

Подготовка упражнения

Прежде чем заняться исследованием того, как ресурсы управления ролевым доступом RBAC влияют на то, что можно делать через сервер API, необходимо убедиться, что плагин RBAC активирован в кластере. Прежде всего убедитесь, что вы используете, по крайней мере, версию 1.6 Kubernetes и что плагин RBAC является единственным сконфигурированным плагином авторизации. Параллельно может быть активировано несколько других плагинов, и если один из них позволяет выполнить действие, то действие разрешается.

ПРИМЕЧАНИЕ. Если вы используете GKE 1.6 или 1.7, вам нужно явным образом деактивировать устаревшую авторизацию, создав кластер с параметром `--no-enable-legacy-authorization`. Если же вы используете Minikube, то вам также может потребоваться активировать плагин RBAC, запустив Minikube с параметром `--extra-config=apiserver.Authorization.Mode=RBAC`.

Если вы следовали инструкциям о том, как деактивировать RBAC в главе 8, то самое время снова его активировать, выполнив следующую ниже команду:

```
$ kubectl delete clusterrolebinding permissive-binding
```

Для того чтобы опробовать управление ролевым доступом RBAC, вы запустите модуль, через который попытаетесь обменяться с сервером API, как вы это делали в главе 8. Но на этот раз вы запустите два модуля в разных пространствах имен, чтобы увидеть, как работает обеспечение безопасности в соответствии с пространством имен.

В примерах из главы 8 вы запускали два контейнера, для того чтобы продемонстрировать, как приложение в одном контейнере использует другой контейнер, с целью взаимодействия с сервером API. На этот раз вы запустите один контейнер (на основе образа `kubectl-proxy`) и примените команду `kubectl exec` для прямого запуска утилиты `curl` внутри этого контейнера. Прокси позаботится об аутентификации и HTTPS, поэтому вы сможете сосредоточиться на авторизационном аспекте обеспечения безопасности сервера API.

Создание пространств имен и запуск модулей

Вы создадите один модуль в пространстве имен `foo`, а другой – в пространстве имен `bar`, как показано в следующем ниже листинге.

Листинг 12.7. Запуск тестовых модулей в разных пространствах имен

```
$ kubectl create ns foo
namespace "foo" created
$ kubectl run test --image=luksa/kubectl-proxy -n foo
deployment "test" created
$ kubectl create ns bar
namespace "bar" created
$ kubectl run test --image=luksa/kubectl-proxy -n bar
deployment "test" created
```

Теперь откройте два терминала и примените `kubectl exec` для запуска оболочки внутри каждого из двух модулей (по одному в каждом терминале). Например, чтобы запустить оболочку в модуле в пространстве имен `foo`, сначала получите имя модуля:

```
$ kubectl get po -n foo
NAME                   READY  STATUS   RESTARTS  AGE
test-145485760-ttq36   1/1    Running  0         1m
```

Затем используйте это имя в команде `kubectl exec`:

```
$ kubectl exec -it test-145485760-ttq36 -n foo sh
/ #
```

Сделайте то же самое и в другом терминале, но для модуля в пространстве имен `bar`.

Вывод списка служб из ваших модулей

Для того чтобы удостовериться, что управление ролевым доступом RBAC включено и предотвращает чтение модулем состояния кластера, используйте утилиту `curl` для вывода списка служб в пространстве имен `foo`:

```
/ # curl localhost:8001/api/v1/namespaces/foo/services
User "system:serviceaccount:foo:default" cannot list services in the
namespace "foo".
```

Вы подключаетесь к `localhost: 8001`; именно там процесс `kubectl proxy` прослушивает (как описано в главе 8). Данный процесс получил ваш запрос и отправил его на сервер API, аутентифицируясь как учетная запись ServiceAccount, действующая по умолчанию, в пространстве имен `foo` (как видно из отклика сервера API).

Сервер API откликнулся тем, что учетной записи службы не разрешено выводить список служб в пространстве имен `foo`, несмотря на то что модуль работает в том же пространстве имен. Вы увидели управление ролевым доступом RBAC в действии. Разрешения по умолчанию для учетной записи ServiceAccount не позволяют ей выводить список ресурсов или их изменять. Теперь давайте узнаем, как разрешить учетной записи это сделать. Сначала вам потребуется создать ресурс Role.

12.2.3 Использование ролей и привязок ролей

Ресурс Role определяет, какие действия можно предпринимать и на каких ресурсах (или, как отмечалось ранее, какие типы HTTP-запросов можно выполнять и на каких ресурсах RESTful). В следующем ниже листинге определяется роль, которая позволяет пользователям получать и выводить список служб в пространстве имен `foo`.

Листинг 12.8. Определение роли: service-reader.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: foo
  name: service-reader
rules:
- apiGroups: [""]
  verbs: ["get", "list"]
  resources: ["services"]
```

Роли имеют пространства имен (если пространство имен опущено, то используется текущее пространство имен)

Службы – это ресурсы в ключевой группе apiGroup, которая не имеет имени – отсюда и ""

Получение отдельных служб (по имени) и вывод списка всех служб разрешены

Это правило относится к службам (имя должно использоваться во множественном числе!)

ПРЕДУПРЕЖДЕНИЕ. При указании ресурсов необходимо использовать форму множественного числа.

Данный ресурс Role будет создан в пространстве имен `foo`. В главе 8 вы узнали, что каждый тип ресурса принадлежит к группе API, которую вы указываете в манифесте ресурса в поле `apiVersion` (вместе с версией). В определении роли необходимо указывать группу `apiGroup` для ресурсов, перечисленных в каждом включенном в определение правиле. Если вы разрешаете доступ к ресурсам, принадлежащим разным группам API, вы используете несколько правил.

ПРИМЕЧАНИЕ. В данном примере вы предоставляете доступ ко всем ресурсам службы, но при этом вы также можете разрешать доступ только к конкретным экземплярам служб, указав их имена через дополнительное поле `resourceNames`.

Рисунок 12.4 показывает роль, ее глаголы и ресурсы, а также пространство, в котором она будет создана.

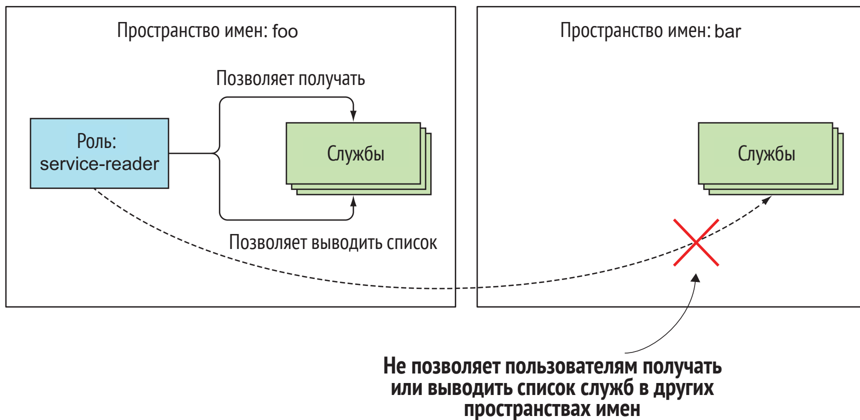


Рис. 12.4. Роль service-reader позволяет получать и выводить список служб в пространстве имен foo

Создание роли

Создайте предыдущую роль в пространстве имен `foo`:

```
$ kubectl create -f service-reader.yaml -n foo
role "service-reader" created
```

ПРИМЕЧАНИЕ. Параметр `-n` является аббревиатурой для параметра `--namespace`.

Обратите внимание, что если вы используете GKE, то предыдущая команда может не сработать, так как у вас нет прав администратора кластера. Чтобы предоставить эти права, выполните следующую ниже команду:

```
$ kubectl create clusterrolebinding cluster-admin-binding
➥   --clusterrole=cluster-admin --user=your.email@address.com
```

Вместо создания роли `service-reader` из файла YAML вы также можете ее создать с помощью специальной команды `kubectl create role`. Давайте воспользуемся этим методом для создания роли в пространстве имен `bar`:

```
$ kubectl create role service-reader --verb=get --verb=list
➥   --resource=services -n bar
role "service-reader" created
```

Эти две роли позволяют выводить список служб в пространствах имен `foo` и `bar` из двух модулей (работающих соответственно в пространствах имен `foo` и `bar`). Но создание двух ролей недостаточно (вы можете проверить, выполнив команду `curl` еще раз). Вам нужно привязать каждую из ролей к учетным записям ServiceAccount в соответствующих пространствах имен.

Привязка роли к учетной записи службы

Роль определяет, какие действия могут выполняться, но не определяет, кто может их выполнять. Для этого необходимо привязать роль к субъекту, которым может быть пользователь, учетная запись службы или группа (пользователей или учетных записей служб).

Привязка ролей к субъектам достигается путем создания ресурса привязки роли, RoleBinding. Для того чтобы привязать роль к учетной записи службы по умолчанию, выполните следующую ниже команду:

```
$ kubectl create rolebinding test --role=service-reader
➥   --serviceaccount=foo:default -n foo
rolebinding "test" created
```

Данная команда должна быть понятной. Вы создаете привязку роли RoleBinding, которая привязывает роль `service-reader` к учетной записи по умолчанию в пространстве имен `foo`. Вы создаете привязку роли RoleBinding в пространстве имен `foo`. Привязка роли и указанная учетная запись и роль показаны на рис. 12.5.

> **ПРИМЕЧАНИЕ.** Для того чтобы привязать роль к пользователю, а не к учетной записи службы, используйте параметр `--user`, указав имя пользователя. Для того чтобы привязать ее к группе, примените параметр `--group`.

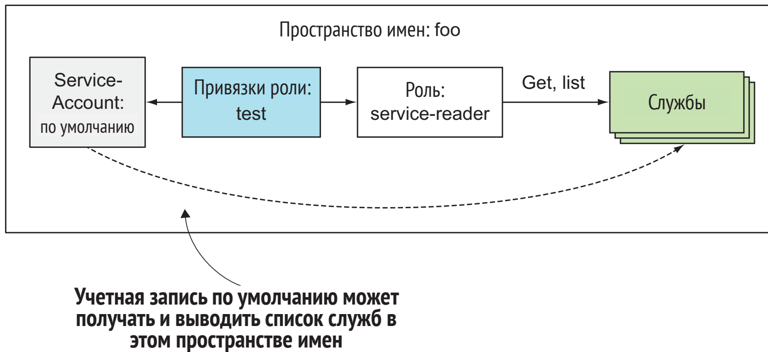


Рис. 12.5. Привязка роли test позволяет привязать учетную запись default к роли service-reader

Следующий ниже листинг показывает YAML созданной вами привязки роли.

Листинг 12.9. Привязка роли RoleBinding, ссылающаяся на роль

```
$ kubectl get rolebinding test -n foo -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: test
  namespace: foo
  ...
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role                    ◁ Эта привязка роли ссылается
  name: service-reader            на роль service-reader
subjects:
- kind: ServiceAccount
  name: default                 ◁ И привязывает ее к учетной записи
  namespace: foo                  default в пространстве имен foo
```

Как вы можете видеть, привязка роли всегда ссылается на одну роль (как явствует из свойства `roleRef`), но может привязывать роли к нескольким субъектам `subjects` (например, одной или нескольким учетным записям и любым количествам пользователей или групп). Поскольку эта привязка роли привязывает роль к учетной записи в пространстве имен `foo`, под которым работает модуль, теперь вы можете вывести список служб из этого модуля.

Листинг 12.10. Получение служб с сервера API

```
/ # curl localhost:8001/api/v1/namespaces/foo/services
{
  "kind": "ServiceList",
  "apiVersion": "v1",
```

```
  "metadata": {
    "selfLink": "/api/v1/namespaces/foo/services",
    "resourceVersion": "24906"
  },
  "items": []      <-- Список элементов пуст, так как службы отсутствуют
}
```

Включение в привязку роли учетных записей из других пространств имен

Модуль в пространстве имен `bar` не может выводить список служб в своем собственном пространстве имен и, безусловно, также не может это делать в пространстве имен `foo`. Но вы можете изменить свои привязки RoleBinding в пространстве имен `foo` и добавить учетную запись ServiceAccount другого модуля, даже если он находится в другом пространстве имен. Выполните следующую ниже команду:

```
$ kubectl edit rolebinding test -n foo
```

Затем добавьте в список субъектов `subjects` указанные ниже строки, как показано в следующем ниже листинге.

Листинг 12.11. Ссылка на учетную запись ServiceAccount из другого пространства имен

```
subjects:
- kind: ServiceAccount
  name: default
  namespace: bar         <-- Вы ссылаетесь на учетную запись, действующую по умолчанию, в пространстве имен bar
```

Теперь вы также можете вывести список служб в пространстве имен `foo` из модуля, работающего в пространстве имен `bar`. Выполните ту же команду, что и в листинге 12.10, но в другом терминале, где вы зашли в оболочку в другом модуле.

Прежде чем перейти к кластерным ролям и привязкам кластерных ролей, давайте подведем итог относительно того, какие ресурсы RBAC у вас есть в настоящее время. У вас есть привязка роли в пространстве имен `foo`, которая ссылается на роль `service-reader` (тоже в пространстве имен `foo`) и привязывает учетные записи ServiceAccount `default` в обоих пространствах имен – `foo` и `bar`, как показано на рис. 12.6.

12.2.4 Применение кластерных ролей (ClusterRole) и привязок кластерных ролей (ClusterRoleBinding)

Роли Role и привязки ролей RoleBinding – это ресурсы, которые имеют пространства имен, то есть они расположены и применяются к ресурсам в од-

ном пространстве имен, но, как мы убедились, привязки ролей также могут ссылаться на учетные записи служб из других пространств имен.

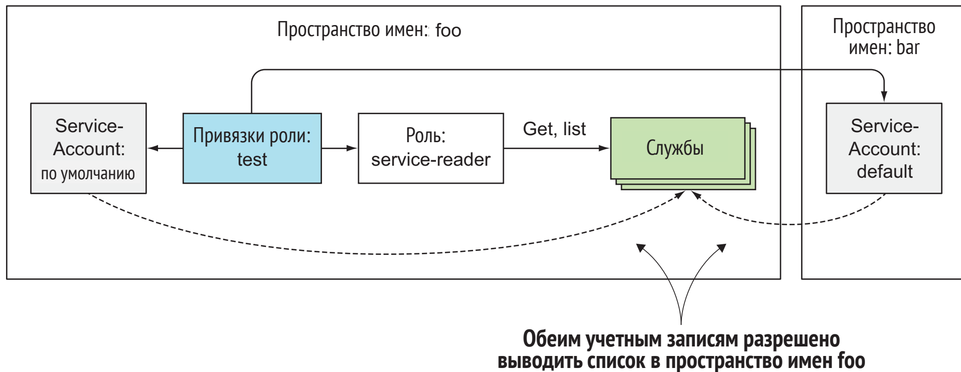


Рис. 12.6. Привязка роли привязывает учетные записи из разных пространств имен к одной роли

В дополнение к этим ресурсам с пространствами имен также существуют два ресурса RBAC уровня кластера: ClusterRole и ClusterRoleBinding, которые не имеют пространств имен. Давайте посмотрим, зачем они вам нужны.

Обычная роль разрешает доступ к ресурсам, находящимся только в том же пространстве имен, в котором находится роль. Если вы хотите предоставить кому-то доступ к ресурсам в разных пространствах имен, то вам нужно создать роль и привязку в каждом из этих пространств имен. Если вы хотите распространить это действие на все пространства имен (возможно, это потребуется администратору кластера), то вам нужно создать одну и ту же роль и привязку в каждом пространстве имен. При создании дополнительного пространства имен вы должны помнить также и о том, чтобы создать два ресурса.

Как вы узнали из книги, некоторые ресурсы вообще не имеют пространств имен (включая узлы Node, постоянные тома PersistentVolume, пространства имен Namespace и т. д.). Мы также отметили, что сервер API предоставляет доступ к некоторым URL-путям, которых не представляют ресурсы (в частности, `/healthz`). Обычные роли не могут предоставлять доступ к этим ресурсам или нересурсным URL-путям. Это могут делать кластерные роли.

Кластерная роль ClusterRole – это ресурс кластерного уровня, предоставляющий доступ к ресурсам без пространств имен или нересурсным URL-путям, или используемый в качестве общей роли для привязки внутри отдельных пространств имен, избавляя вас от необходимости переопределять одну и ту же роль в каждом из них.

Разрешение доступа к ресурсам уровня кластера

Как уже отмечалось, кластерная роль ClusterRole может использоваться для предоставления доступа к ресурсам уровня кластера. Давайте посмотрим, как позволить вашему модулю вывести список постоянных томов PersistentVolume в вашем кластере. Сначала вы создадите кластерную роль с именем `pv-reader`:

```
$ kubectl create clusterrole pv-reader --verb=get,list
➥   --resource=persistentvolumes
clusterrole "pv-reader" created
```

YAML кластерной роли показан в следующем ниже листинге.

Листинг 12.12. Определение кластерной роли

```
$ kubectl get clusterrole pv-reader -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pv-reader
  resourceVersion: "39932"
  selfLink: ...
  uid: e9ac1099-30e2-11e7-955c-080027e6b159
rules:
- apiGroups:
  - ""
  resources:
  - persistentvolumes
  verbs:
  - get
 - list
```

Кластерные роли не имеют пространства имен, следовательно, поле namespace отсутствует

В этом случае правила в точности такие же, как и в обычной роли

Прежде чем вы привяжете эту кластерную роль к учетной записи ServiceAccount вашего модуля, проверьте, может ли модуль выводить список постоянных томов. Выполните следующую ниже команду в первом терминале, где вы выполняете оболочку внутри модуля в пространстве имен `foo`:

```
/ # curl localhost:8001/api/v1/persistentvolumes
User "system:serviceaccount:foo:default" cannot list persistentvolumes at the
cluster scope.
```

> **ПРИМЕЧАНИЕ.** URL-адрес не содержит пространства имен, потому что постоянные тома PersistentVolume не имеют пространства имен.

Как и ожидалось, учетная запись службы ServiceAccount по умолчанию не может выводить список постоянных томов. Для того чтобы позволить ей это делать, вам нужно привязать кластерную роль к вашей учетной записи. Кластерные роли могут быть привязаны к субъектам с обычными привязками ролей, и поэтому вы сейчас создадите привязку роли RoleBinding:

```
$ kubectl create rolebinding pv-test --clusterrole=pv-reader
➥   --serviceaccount=foo:default -n foo
rolebinding "pv-test" created
```

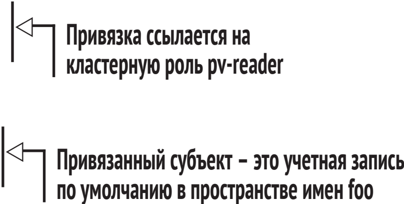
Можно ли сейчас вывести список постоянных томов?

```
/ # curl localhost:8001/api/v1/persistentvolumes
User "system:serviceaccount:foo:default" cannot list persistentvolumes at the
cluster scope.
```

Хм, очень странно. Давайте рассмотрим YAML привязки роли в следующем ниже листинге. Можете ли вы сказать, что в нем не так (если вообще есть что-то)?

Листинг 12.13. Привязка роли, ссылающаяся на кластерную роль

```
$ kubectl get rolebindings pv-test -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pv-test
  namespace: foo
  ...
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pv-reader
subjects:
- kind: ServiceAccount
  name: default
  namespace: foo
```



YAML выглядит прекрасно. Вы ссылаетесь на правильную кластерную роль и правильную учетную запись, как показано на рис. 12.7, так что же тогда не так?

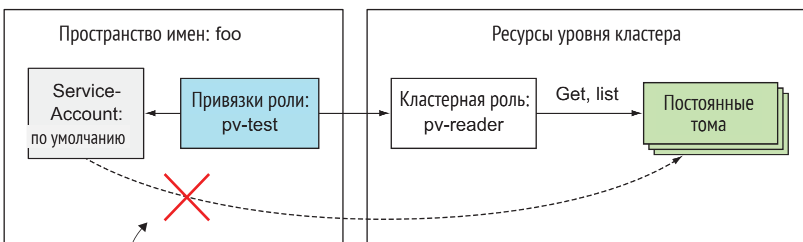


Рис. 12.7. Привязка роли, ссылающаяся на кластерную роль, не предоставляет доступа к ресурсам уровня кластера

Хотя вы можете создать привязку RoleBinding и заставить ее ссылаться на кластерную роль ClusterRole, если требуется разрешить доступ к ресурсам с пространством имен, тот же подход нельзя использовать для ресурсов уровня кластера (без пространства имен). Для того чтобы предоставить доступ к ре-

сурсам уровня кластера, вы всегда должны использовать привязку кластерной роли ClusterRoleBinding.

К счастью, создание привязки ClusterRoleBinding не отличается от создания привязки RoleBinding, но сначала вы очистите и удалите привязку роли RoleBinding:

```
$ kubectl delete rolebinding pv-test
rolebinding "pv-test" deleted
```

Теперь создайте привязку кластерной роли:

```
$ kubectl create clusterrolebinding pv-test --clusterrole=pv-reader
➥   --serviceaccount=foo:default
clusterrolebinding "pv-test" created
```

Как вы можете видеть, вы заменили в команде `rolebinding` на `clusterrolebinding` и не указали (вам не нужно это делать) пространство имен. Рисунок 12.8 показывает, что вы имеете в результате.

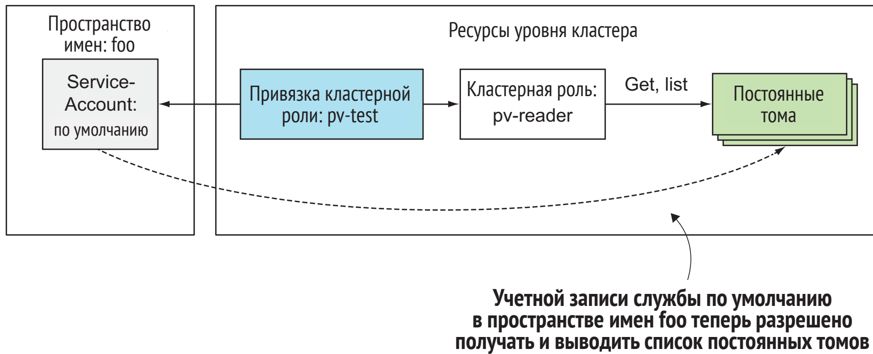


Рис. 12.8. Для предоставления доступа к ресурсам уровня кластера должны использоваться привязка кластерной роли и кластерная роль

Давайте посмотрим, сможете ли вы сейчас вывести список постоянных томов:

```
/ # curl localhost:8001/api/v1/persistentvolumes
{
  "kind": "PersistentVolumeList",
  "apiVersion": "v1",
...
```

Получилось! Оказывается, что при предоставлении доступа к ресурсам уровня кластера необходимо использовать кластерную роль и привязку кластерной роли.

> **СОВЕТ.** Помните, что привязка роли не может предоставлять доступ к ресурсам уровня кластера, даже если она ссылается на привязку кластерной роли.

Предоставление доступа к нересурсным URL-путям

Мы уже отмечали, что сервер API также предоставляет URL-пути, не связанные с ресурсами. Доступ к этим URL-путям также должен быть предоставлен явным образом; в противном случае сервер API отклонит запрос клиента. Как правило, это делается автоматически через кластерную роль `system:discovery` и одноименную привязку кластерной роли, которые появляются среди других предопределенных кластерных ролей и привязок кластерных ролей (мы рассмотрим их в разделе 12.2.5).

Давайте проинспектируем кластерную роль `system:discovery`, показанную в следующем ниже листинге.

Листинг 12.14. Кластерная роль по умолчанию system:discovery

```
$ kubectl get clusterrole system:discovery -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:discovery
  ...
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /swaggerapi
  - /swaggerapi/*
  - /version
verbs:
- get
```

Вместо ссылки на ресурсы это правило ссылается на нересурсные URL-пути

Для этих URL-путей разрешен только метод HTTP GET

Вы можете видеть, что эта кластерная роль ссылается не на ресурсы, а на URL-пути (вместо поля `resources` используется поле `nonResourceURLs`). Для этих URL-адресов поле `verbs` позволяет использовать только метод HTTP `GET`.

> **ПРИМЕЧАНИЕ.** Для нересурсных URL-путей вместо глаголов `create` или `update` используются простые команды HTTP, такие как `post`, `put` и `patch`. Глаголы должны быть указаны в нижнем регистре.

Как и с ресурсами кластерного уровня, кластерные роли для нересурсных URL-путей должны быть привязаны к привязке ClusterRoleBinding. Привязывание их к привязке RoleBinding не будет иметь никакого эффекта. Кластерная роль `system:discovery` имеет соответствующую привязку кластерной роли `system:discovery`, поэтому давайте посмотрим, что в ней находится, исследовав следующий ниже листинг.

Листинг 12.15. Привязка ClusterRoleBinding system:discovery, существующая по умолчанию

```
$ kubectl get clusterrolebinding system:discovery -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:discovery
  ...
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole                          ◁ Эта привязка ClusterRoleBinding ссылается
  name: system:discovery                       на кластерную роль system: discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group                                ◁ Она привязывает кластерную роль ко всем
  name: system:authenticated                   аутентифицированным и неаутентифицированным
- apiGroup: rbac.authorization.k8s.io          пользователям (то есть ко всем)
  kind: Group                                ◁
  name: system:unauthenticated
```

Как и ожидалось, YAML показывает, что привязка ClusterRoleBinding ссылается на кластерную роль `system:discovery`. Она привязала ее к двум группам, `system:authenticated` и `system:unauthenticated`, то есть привязала ко всем пользователям. Это означает, что доступ к URL-путям, перечисленным в кластерной роли, может получать абсолютно каждый.

> **ПРИМЕЧАНИЕ.** Группы находятся в введении плагина аутентификации. Когда запрос получен сервером API, он вызывает плагин аутентификации, для того чтобы получить список групп, к которым принадлежит пользователь. Эта информация затем используется при авторизации.

Вы можете это подтвердить, обратившись к URL-пути `/api` из модуля (через `kubectl proxy`, то есть вы будете аутентифицированы как учетная запись модуля) и из вашей локальной машины, без указания каких-либо токенов аутентификации (что делает вас неаутентифицированным пользователем):

```
$ curl https://$(minikube ip):8443/api -k
{
  "kind": "APIVersions",
  "versions": [
  ...
```

Сейчас вы использовали кластерные роли и привязки кластерных ролей для предоставления доступа к ресурсам уровня кластера и нересурсным URL-путям. Теперь давайте посмотрим, как кластерные роли можно использовать с привязками RoleBinding, имеющими пространства имен, для предоставления

доступа к ресурсам, имеющим пространства имен, в пространстве имен привязки RoleBinding.

Использование кластерных ролей для предоставления доступа к ресурсам в специфических пространствах имен

Кластерные роли не всегда должны быть привязаны к привязкам кластерных ролей уровня кластера. Они также могут быть привязаны к обычным привязкам ролей с пространствами имен. Вы уже начали рассматривать предопределенные кластерные роли, поэтому давайте проанализируем еще одну под названием `view`, которая показана в следующем ниже листинге.

Листинг 12.16. Кластерная роль view, существующая по умолчанию

```
$ kubectl get clusterrole view -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: view
  ...
rules:
- apiGroups:
  - ""
  resources:
  - configmaps
  - endpoints
  - persistentvolumeclaims
  - pods
  - replicationcontrollers
  - replicationcontrollers/scale
  - serviceaccounts
  - services
  verbs:
  - get
  - list
  - watch
...
```

Данное правило применяется к этим ресурсам (примечание: все они являются ресурсами с пространствами имен)

Как следует из названия кластерной роли, она разрешает только чтение перечисленных ресурсов, не запись

Данная кластерная роль имеет много правил. В листинге показано только первое. Это правило позволяет получать, выводить список и наблюдать за такими ресурсами, как словари конфигурации ConfigMap, конечные точки Endpoints, заявки на получение постоянных томов PersistentVolumeClaim и т. д. Эти ресурсы имеют пространства имен, даже если вы рассматриваете кластерную роль ClusterRole (не обычную роль с пространством имен). Что именно делает эта кластерная роль?

Это зависит от того, привязана ли она к привязке ClusterRoleBinding или же привязке RoleBinding (она может быть привязана с любой). Если вы создаете

привязку ClusterRoleBinding и ссылаетесь в ней на кластерную роль, субъекты, перечисленные в этой привязке, могут просматривать указанные ресурсы во всех пространствах имен. Если, с другой стороны, вы создаете привязку RoleBinding, то субъекты, перечисленные в этой привязке, могут просматривать ресурсы только в пространстве имен привязки RoleBinding. Сейчас вы попробуете оба варианта.

Вы увидите, как эти два параметра влияют на способность тестового модуля выводить список модулей. Сначала давайте посмотрим, что происходит до того, как установлены какие-либо привязки:

```
/ # curl localhost:8001/api/v1/pods
User "system:serviceaccount:foo:default" cannot list pods at the cluster scope./ #
/ # curl localhost:8001/api/v1/namespaces/foo/pods
User "system:serviceaccount:foo:default" cannot list pods in the namespace "foo".
```

С помощью первой команды вы пытаетесь вывести список модулей во всех пространствах имен. С помощью второй вы пытаетесь вывести список модулей в пространстве имен `foo`. В обоих случаях сервер не позволяет вам это делать.

Теперь давайте посмотрим, что происходит, когда вы создаете привязку ClusterRoleBinding и привязываете ее к учетной записи модуля:

```
$ kubectl create clusterrolebinding view-test --clusterrole=view
➥   --serviceaccount=foo:default
clusterrolebinding "view-test" created
```

Сможет ли модуль теперь вывести список модулей в пространстве имен `foo`?

```
/ # curl localhost:8001/api/v1/namespaces/foo/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
```

Может! Поскольку вы создали привязку ClusterRoleBinding, она применяется ко всем пространствам имен. Модуль в пространство имен `foo` может равным образом выводить список модулей в пространстве имен `bar`:

```
/ # curl localhost:8001/api/v1/namespaces/bar/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
```

Хорошо. Модулю разрешено выводить список модулей в другом пространстве имен. Он также может получать модули во всех пространствах имен, попав на URL-путь /api/v1/pods:

```
/ # curl localhost:8001/api/v1/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
```

Как и ожидалось, модуль может получить список всех модулей в кластере. Подытоживая: объединение привязки ClusterRoleBinding с кластерной ролью, ссылающейся на ресурсы с пространством имен, позволяет модулю получать доступ к ресурсам с пространством имен в любом пространстве имен, как показано на рис. 12.9.

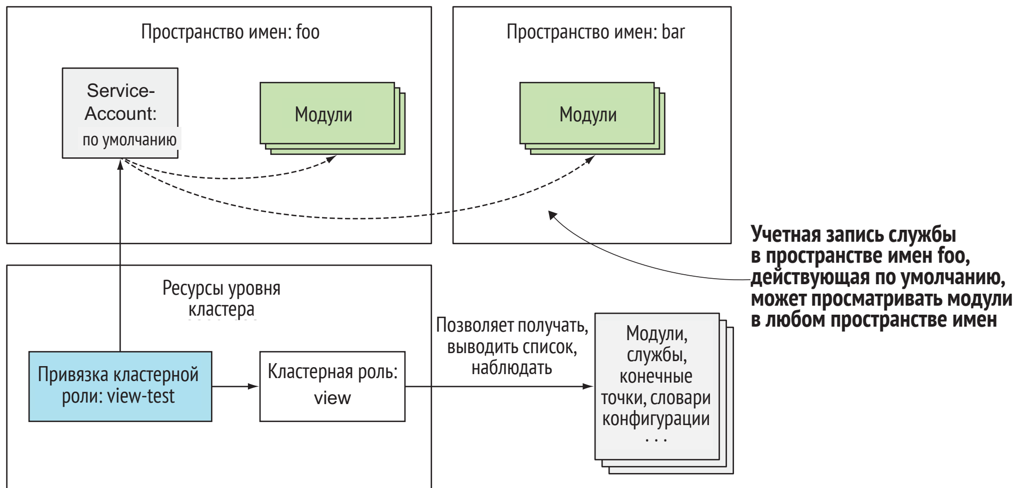


Рис. 12.9. Привязка ClusterRoleBinding и кластерная роль ClusterRole предоставляют разрешение на ресурсы во всех пространствах имен

Теперь давайте посмотрим, что произойдет, если заменить привязку ClusterRoleBinding обычной привязкой RoleBinding. Сначала удалим привязку ClusterRoleBinding:

```
$ kubectl delete clusterrolebinding view-test
clusterrolebinding "view-test" deleted
```

Затем создадим привязку RoleBinding. Поскольку привязка RoleBinding имеет пространство имен, необходимо указать пространство имен, в котором ее нужно создать. Создайте ее в пространстве имен `foo`:

```
$ kubectl create rolebinding view-test --clusterrole=view
➥   --serviceaccount=foo:default -n foo
rolebinding "view-test" created
```

Теперь у вас есть привязка RoleBinding в пространстве имен `foo`, привязывающая учетную запись `default` в том же пространстве имен к кластерной роли ClusterRole `view`. К чему ваш модуль теперь может иметь доступ?

```
/ # curl localhost:8001/api/v1/namespaces/foo/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
/ # curl localhost:8001/api/v1/namespaces/bar/pods
User "system:serviceaccount:foo:default" cannot list pods in the namespace "bar".
/ # curl localhost:8001/api/v1/pods
User "system:serviceaccount:foo:default" cannot list pods at the cluster scope.
```

Как вы можете видеть, ваш модуль может выводить список модулей в пространстве имен `foo`, но не в любом другом конкретном пространстве имен или во всех пространствах имен. Это показано на рис. 12.10.

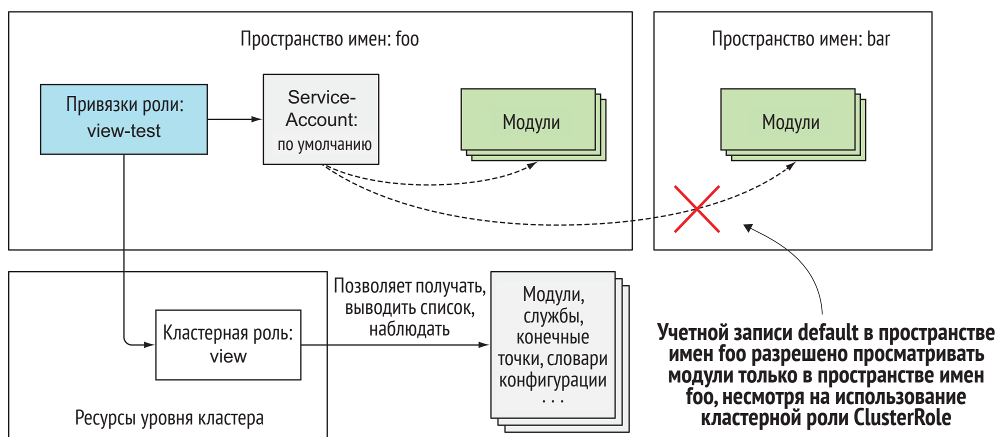


Рис. 12.10. Привязка RoleBinding, ссылающаяся на кластерную роль ClusterRole, предоставляет доступ только к ресурсам в пространстве имен привязки RoleBinding

Резюмирование комбинаций роли, кластерной роли, привязки роли и привязки кластерной роли

Мы рассмотрели много различных комбинаций, и вам, возможно, трудно запомнить, когда использовать каждую из них. Давайте попробуем разобраться во всех этих комбинациях, разбив их на категории в соответствии с конкретным случаем использования. См. табл. 12.2.

Таблица 12.2. Когда следует использовать конкретные комбинации типов ролей и привязок

Для доступа	Используемый тип роли	Используемый тип привязки
Ресурсы уровня кластера (узлы, постоянные тома и прочие)	`ClusterRole`	`ClusterRoleBinding`
Нересурсные URL-пути (`/api`, `/healthz` и прочие)	`ClusterRole`	`ClusterRoleBinding`

Для доступа	Используемый тип роли	Используемый тип привязки
Ресурсы с пространством имен в любом пространстве имен (и во всех пространствах имен)	`ClusterRole`	`ClusterRoleBinding`
Ресурсы с пространством имен в специфическом пространстве имен (просторное использование одной кластерной роли во множестве пространств имен)	`ClusterRole`	`RoleBinding`
Ресурсы с пространством имен в специфическом пространстве имен (роль должна быть определена в каждом пространстве имен)	`Role`	`RoleBinding`

Надеюсь, теперь связи между четырьмя ресурсами управления ролевым доступом RBAC стали гораздо яснее. Не переживайте, если вы по-прежнему чувствуете, что еще не все поняли. Все может проясниться, когда мы займемся исследованием предварительно настроенных кластерных ролей и привязок кластерных ролей в следующем разделе.

12.2.5 Кластерные роли и привязки кластерных ролей, существующие по умолчанию

Kubernetes поставляется со стандартным набором кластерных ролей и привязок кластерных ролей, которые обновляются каждый раз, когда сервер API запускается. Это гарантирует, что все роли и привязки, существующие по умолчанию, будут созданы заново, если вы их по ошибке удалите или если более новая версия Kubernetes использует другую конфигурацию кластерных ролей и привязок.

Кластерные роли и привязки, существующие по умолчанию, приведены в следующем ниже листинге.

Листинг 12.17. Вывод списка всех привязок ClusterRoleBinding и ролей ClusterRole

```
$ kubectl get clusterrolebindings
NAME                                          AGE
cluster-admin                                 1d
system:basic-user                             1d
system:controller:attachdetach-controller    1d
...
system:controller:ttl-controller             1d
system:discovery                              1d
system:kube-controller-manager                1d
system:kube-dns                               1d
system:kube-scheduler                         1d
```

```
system:node                                 1d
system:node-proxier                         1d

$ kubectl get clusterroles
NAME                                        AGE
admin                                       1d
cluster-admin                               1d
edit                                        1d
system:auth-delegator                       1d
system:basic-user                           1d
system:controller:attachdetach-controller   1d
...
system:controller:ttl-controller            1d
system:discovery                            1d
system:heapster                             1d
system:kube-aggregator                      1d
system:kube-controller-manager              1d
system:kube-dns                             1d
system:kube-scheduler                       1d
system:node                                 1d
system:node-bootstrapper                    1d
system:node-problem-detector                1d
system:node-proxier                         1d
system:persistent-volume-provisioner        1d
view                                        1d
```

Самыми важными ролями являются кластерные роли `view`, `edit`, `admin` и `cluster-admin`. Они предназначены для привязки к учетным записям ServiceAccount, используемым в пользовательских модулях.

Разрешение ресурсам доступа в режиме только для чтения с помощью кластерной роли view

В предыдущем примере вы уже использовали существующую по умолчанию кластерную роль `view`. Она позволяет читать большинство ресурсов в пространстве имен, за исключением ролей, привязок ролей и секретов. Вам, наверное, интересно, почему не секреты? Потому что один из этих секретов может включать токен аутентификации с бóльшими привилегиями, чем те, которые определены в кластерной роли `view`, и может позволить пользователю маскироваться под другого пользователя, чтобы получать дополнительные привилегии (эскалацию привилегий).

Разрешение на внесение изменений в ресурсы с помощью кластерной роли edit

Далее идет кластерная роль `edit`, которая позволяет вносить изменения в ресурсы в пространстве имен, но также позволяет и чтение, и модификацию секретов. Однако она не позволяет просматривать или изменять роли или

привязки ролей – опять же, это необходимо для предотвращения эскалации привилегий.

Предоставление полного контроля над пространством имен с помощью кластерной роли admin

Полный контроль над ресурсами в пространстве имен предоставляется в кластерной роли `admin`. Субъекты с этой кластерной ролью могут читать и вносить изменения в любой ресурс в пространстве имен, кроме квот ResourceQuota (мы узнаем, что это такое, в главе 14) и самого ресурса пространства имен Namespace. Основное различие между кластерными ролями `edit` и `admin` заключается в возможности просмотра и внесении изменения в роли и привязки ролей в пространстве имен.

> **ПРИМЕЧАНИЕ.** Для того чтобы предотвратить эскалацию привилегий, сервер API позволяет пользователям создавать и обновлять роли, только если они уже имеют все разрешения, перечисленные в этой роли (и для той же области действия).

Предоставление полного контроля с помощью кластерной роли cluster-admin

Полный контроль над кластером Kubernetes может быть предоставлен путем присвоения субъекту кластерной роли `cluster-admin`. Как вы видели раньше, кластерная роль `admin` не позволяет пользователям вносить изменения в объекты ResourceQuota конкретного пространства имен или в сами ресурсы Namespace. Если вы хотите разрешить пользователю это делать, то вам нужно создать привязку RoleBinding, которая ссылается на кластерную роль `cluster-admin`. Это дает пользователю, включенному в привязку RoleBinding, полный контроль над всеми аспектами пространства имен, в котором создана привязка RoleBinding.

Если вы обратили внимание, то вы, вероятно, уже знаете, как предоставлять пользователям полный контроль над всеми пространствами имен в кластере. Совершенно верно, путем ссылки на кластерную роль `cluster-admin` в привязке ClusterRoleBinding, а не в привязке RoleBinding.

Другие кластерные роли, установленные по умолчанию

Список кластерных ролей, установленных по умолчанию, включает в себя большое количество других кластерных ролей, которые начинаются с префикса `system:`. Они предназначены для использования различными компонентами Kubernetes. Среди них вы найдете такие роли, как `system:kube-scheduler`, которая, очевидно, используется планировщиком, `system:node`, которая используется агентом Kubelets, и т. д.

Хотя менеджер контроллеров работает как один модуль, каждый контроллер, работающий внутри него, может использовать отдельную кластерную роль и привязку кластерной роли (они имеют префикс `system: controller:`).

Каждая из этих системных кластерных ролей имеет соответствующую привязку ClusterRoleBinding, которая привязывает ее к пользователю, которого системный компонент аутентифицирует. Например, привязка ClusterRoleBinding `system:kube-scheduler` назначает одноименную кластерную роль пользователю `system:kube-scheduler`, то есть имени пользователя, под которым планировщик аутентифицируется.

12.2.6 Предоставление разумных авторизационных разрешений

По умолчанию учетная запись службы в пространстве имен не имеет разрешений, кроме разрешений для неаутентифицированного пользователя (как вы помните из одного из предыдущих примеров, кластерная роль `system:discovery` и ассоциированная привязка разрешают любому делать запросы GET к нескольким нересурсным URL-путям). Поэтому модули по умолчанию не могут просматривать даже состояние кластера. Решение о том, предоставлять ли им для этого соответствующие разрешения, остается за вами.

Очевидно, предоставление всем своим учетным записям кластерной роли `cluster-admin` – нехорошая идея. Как и всегда в случае с безопасностью, лучше всего предоставлять всем учетным записям только те разрешения, которые необходимы для выполнения их работы, и ни одного разрешения больше (*принцип наименьшего количества привилегий*).

Создание специфических учетных записей служб для каждого модуля

Хорошей идеей является создание конкретной учетной записи для каждого модуля (или набора реплик модуля) и затем привязка ее к индивидуализированной роли (или кластерной роли) через привязку роли (только не через привязку кластерной роли, потому что это дало бы модулю доступ к ресурсам в других пространствах имен, что, вероятно, не то, чего вы хотите).

Если один из ваших модулей (точнее, работающее в нем приложение) нуждается только в чтении модулей, в то время как другой, кроме того, нуждается во внесении в них изменений, то создайте две разные учетные записи ServiceAccount и дайте этим модулям их использовать, определив свойство `serviceAccountName` в спецификации модуля, как вы узнали в первой части этой главы. Избегайте добавления в учетную запись, действующую по умолчанию, в конкретном пространстве имен всех необходимых разрешений, требуемых обоими модулями.

Ожидайте того, что ваши приложения будут подвергнуты опасности

Ваша цель – уменьшить возможности злоумышленника завладеть вашим кластером. Современные сложные приложения содержат множество уязвимостей. Вы должны ожидать, что нежелательные лица в конечном итоге получат токен аутентификации учетной записи ServiceAccount, поэтому, для того чтобы предотвращать реальный ущерб, вы всегда должны ограничивать учетную запись службы.

12.3 Резюме

Эта глава дала вам основополагающее понимание того, как защищать сервер API Kubernetes. Вы узнали следующее:

- клиентами сервера API могут быть как пользователь-человек, так и работающие в модулях приложения;
- приложения в модулях связаны с учетной записью службы ServiceAccount;
- пользователи и учетные записи связаны с группами;
- по умолчанию модули запускаются под установленной по умолчанию учетной записью, которая автоматически создается для каждого пространства имен;
- дополнительные учетные записи служб могут быть созданы вручную и связаны с модулем;
- учетные записи могут быть сконфигурированы, чтобы разрешать монтирование только ограниченного списка секретов в данном модуле;
- учетная запись также может использоваться для прикрепления к модулям секретов извлечения образов, поэтому вам не нужно указывать секреты в каждом модуле;
- роли и кластерные роли определяют, какие действия могут выполняться на ресурсах;
- привязки ролей RoleBinding и привязки кластерных ролей ClusterRoleBinding привязывают роли и кластерные роли к пользователям, группам и учетным записям;
- каждый кластер поставляется с набором кластерных ролей, действующих по умолчанию, и привязок кластерных ролей.

В следующей главе вы узнаете, как защищать узлы кластера от модулей и как изолировать модули друг от друга путем обеспечения защиты сети.

Глава 13

Защита узлов кластера и сети

Эта глава посвящена:

- использованию в модулях стандартных пространств имен Linux узла;
- запуску контейнеров от разных пользователей;
- запуску привилегированных контейнеров;
- добавлению или удалению возможностей ядра ОС в контейнере;
- определению политик безопасности для ограничения того, что модули могут делать;
- обеспечению безопасности сети модулей.

В предыдущей главе мы говорили о защите сервера API. Если злоумышленник получает доступ к серверу API, он может выполнить все, что угодно, упаковав код в образ контейнера и запустив его в модуле. Но может ли он нанести реальный ущерб? Разве контейнеры не изолированы от других контейнеров и от узла, на котором они работают?

Это не обязательно так. В этой главе вы узнаете, как разрешать модулям доступ к ресурсам узла, на котором они работают. Вы также узнаете, как настраивать кластер таким образом, чтобы пользователи не могли делать со своими модулями все, что они хотят. Затем, в последней части главы, вы также узнаете, как защитить сеть, которую модули используют для взаимодействия.

13.1 Использование в модуле пространств имен хоста

Контейнеры в модуле обычно работают в отдельных пространствах имен Linux, которые изолируют свои процессы от процессов, запущенных в других контейнерах или в стандартных пространствах имен узла.

Например, мы узнали, что каждый модуль получает свое собственное пространство IP-адресов и портов, потому что он использует собственное сетевое

пространство имен. Схожим образом каждый модуль имеет свое собственное дерево процессов, потому что он имеет свое собственное пространство имен PID, а также использует свое собственное пространство имен IPC, позволяя через механизм межпроцессного взаимодействия (IPC) взаимодействовать друг с другом только тем процессам, которые находятся в одном модуле.

13.1.1 Использование в модуле сетевого пространства имен узла

Определенные модули (обычно системные модули) должны работать в стандартных пространствах имен хоста, что позволяет им видеть и управлять ресурсами и устройствами уровня узла. Например, модулю может потребоваться использовать сетевые адаптеры узла вместо собственных виртуальных сетевых адаптеров. Это может быть достигнуто путем присвоения значения `true` свойству `hostNetwork` в секции `spec` модуля.

В этом случае, как показано на рис. 13.1, модуль станет использовать сетевые интерфейсы узла, вместо того чтобы иметь свой собственный набор. Это означает, что модуль не получает свой собственный IP-адрес, и если он выполняет процесс, который привязывается к порту, то этот процесс будет привязан к порту узла.

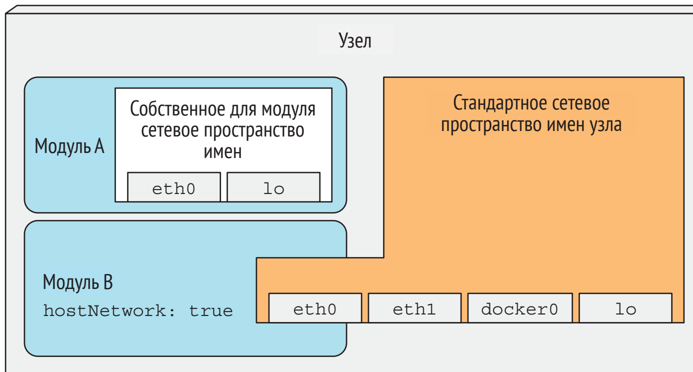


Рис. 13.1. Модуль с hostNetwork: true использует сетевые интерфейсы узла вместо своих собственных

Вы можете попробовать запустить такой модуль. В следующем ниже листинге показан пример манифеста модуля.

Листинг 13.1. Модуль, использующий сетевое пространство имен узла: pod-with-host-network.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-host-network
spec:
```

```
  hostNetwork: true                      <-- Использование сетевого
  containers:                                пространства имен хост-узла
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
```

После запуска этого модуля можно применить следующую ниже команду, которая покажет, что он на самом деле использует сетевое пространство имен хоста (в частности, он видит все сетевые адаптеры узла).

Листинг 13.2. Сетевые интерфейсы в модуле, использующем сетевое пространство имен хоста

```
$ kubectl exec pod-with-host-network ifconfig
docker0   Link encap:Ethernet HWaddr 02:42:14:08:23:47
          inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
          ...
eth0      Link encap:Ethernet HWaddr 08:00:27:F8:FA:4E
inet      addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
          ...
lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          ...
veth1178d4f Link encap:Ethernet HWaddr 1E:03:8D:D6:E1:2C
          inet6 addr: fe80::1c03:8dff:fed6:e12c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          ...
```

Когда компоненты плоскости управления Kubernetes развертываются как модули (например, при развертывании кластера с помощью `kubeadm`, как описано в приложении В), вы обнаружите, что эти модули используют параметр `hostNetwork`, который фактически заставляет их вести себя так, как если бы они не выполнялись внутри модуля.

13.1.2 Привязка к порту хоста без использования сетевого пространства имен хоста

Родственное функциональное средство позволяет модулям привязываться к порту в стандартном пространстве имен узла, но при этом иметь собственное сетевое пространство имен. Это делается с помощью свойства `hostPort` в одном из портов контейнера, определенном в поле `spec.containers.ports`.

Не путайте модули, использующие `hostPort`, с модулями, предоставляемыми через службу `NodePort`. Как показано на рис. 13.2, это две разные вещи.

Первое, что вы заметите на рисунке, – когда модуль использует `hostPort`, подключение к порту узла перенаправляется непосредственно модулю, работающему на этом узле, а в случае со службой `NodePort` подключение к порту узла перенаправляется случайно выбранному модулю (возможно, на другой

узел). Другое различие состоит в том, что в случае с модулями, использующими hostPort, порт узла привязан только к тем узлам, которые выполняют такие модули, тогда как службы NodePort привязывают порт ко всем узлам, даже к тем, которые не выполняют такой модуль (как в случае с узлом 3 на рисунке).

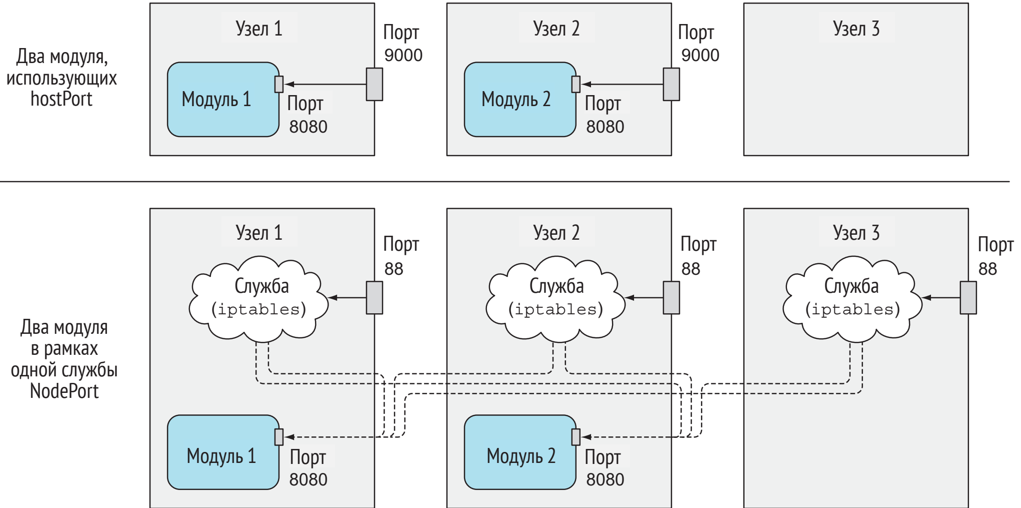


Рис. 13.2. Разница между модулями, использующими hostPort, и модулями позади службы NodePort

Важно понимать, что если модуль использует конкретный порт хоста, то только один экземпляр модуля может быть назначен каждому узлу, потому что два процесса не могут привязываться к одинаковому порту хоста. Планировщик учитывает это при назначении модулей, поэтому он не назначает несколько модулей на один узел. Это показано на рис. 13.3. Если у вас есть три узла и вы хотите развернуть четыре реплики модуля, только три будут назначены (один модуль останется в ожидании).

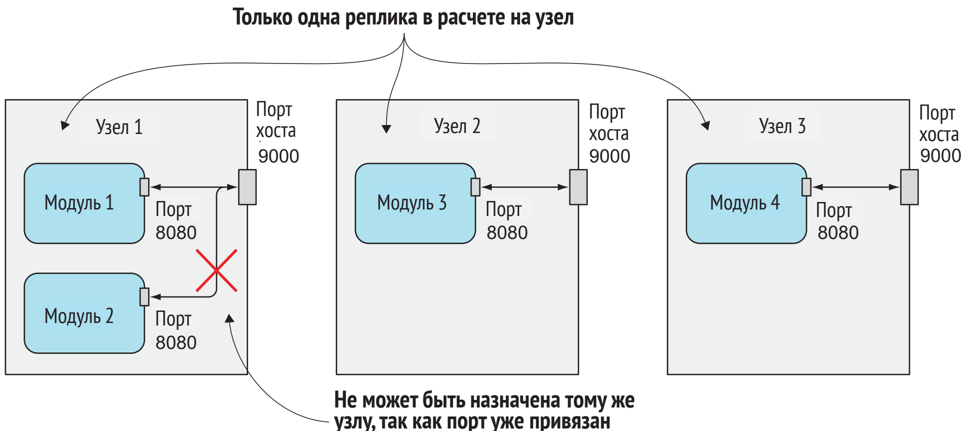


Рис. 13.3. Если используется порт хоста, то узлу может быть назначен только один экземпляр модуля

Давайте посмотрим, как задать `hostPort` в определении YAML модуля. Следующий ниже листинг показывает YAML, который запускает ваш модуль `kubia` и привязывает ее к порту 9000 узла.

Листинг 13.3. Привязка модуля к порту в пространстве портов узла: kubia-hostport.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-hostport
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080     <-- Контейнер можно достичь на порту 8080 IP-адреса модуля
      hostPort: 9000          <-- Он также может быть достигнут на порту 9000 узла, на котором он развернут
      protocol: TCP
```

После создания этого модуля вы можете получить к нему доступ через порт 9000 узла, на который он назначен. Если у вас несколько узлов, то вы увидите, что не сможете получить доступа к данному модулю через этот порт на других узлах.

> **ПРИМЕЧАНИЕ.** Если вы пробуете это на GKE, то вам нужно правильно сконфигурировать брэндмауэр, используя для этого команду `gcloud compute firewall-rules`, как вы это делали в главе 5.

Функциональное средство `hostPort` в основном используется для предоставления системных служб, которые развертываются на каждом узле с помощью наборов демонов DaemonSet. Первоначально оно также использовалось специалистами, чтобы гарантировать, что две реплики одного и того же модуля никогда не будут назначены на один и тот же узел, но теперь для того, чтобы достичь этого, у вас есть более оптимальный способ – он объясняется в главе 16.

13.1.3 Использование пространств имен PID и IPC узла

Свойства `hostPID` и `hostIPC` секции `spec` модуля аналогичны параметру `hostNetwork`. Если задать для них значение `true`, то контейнеры модуля будут использовать пространства имен PID и IPC узла, позволяя процессам, запущенным в контейнерах, соответственно видеть все другие процессы на узле или взаимодействовать с ними через IPC. Соответствующий пример смотрите в следующем ниже листинге.

Листинг 13.4. Использование пространств имен PID и IPC хоста: pod-with-host-pid-and-ipc.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-host-pid-and-ipc
spec:
  hostPID: true        ◁ Вы хотите, чтобы модуль использовал пространство имен PID узла
  hostIPC: true        ◁ Вы также хотите, чтобы модуль использовал пространство имен IPC
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
```

Напомним, что модули обычно видят только свои собственные процессы, но если вы запустите этот модуль, а затем выведете список процессов изнутри его контейнера, то, как показано в следующем ниже листинге, вы увидите все процессы, работающие на узле хоста, а не только те, которые работают в контейнере.

Листинг 13.5. Процессы, видимые в модуле с hostPID: true

```
$ kubectl exec pod-with-host-pid-and-ipc ps aux
PID  USER    TIME   COMMAND
  1  root    0:01   /usr/lib/systemd/systemd --switched-root --system ...
  2  root    0:00   [kthreadd]
  3  root    0:00   [ksoftirqd/0]
  5  root    0:00   [kworker/0:0H]
  6  root    0:00   [kworker/u2:0]
  7  root    0:00   [migration/0]
  8  root    0:00   [rcu_bh]
  9  root    0:00   [rcu_sched]
 10  root    0:00   [watchdog/0]
...
```

Если присвоить свойству `hostIPC` значение `true`, то процессы в контейнерах модуля смогут также взаимодействовать со всеми другими процессами, работающими на узле, используя межпроцессное взаимодействие.

13.2 Конфигурирование контекста безопасности контейнера

Помимо разрешения модулю использовать Linux-пространства имен хоста, на модуле и его контейнере можно также настроить другую фукциональность, связанную с безопасностью. Это делается через свойства `securityContext`, ко-

торые могут быть указаны непосредственно в секции spec модуля и внутри секции spec отдельных контейнеров.

Что конфигурируется в контексте безопасности

Конфигурирование контекста безопасности позволяет вам выполнять различные действия:

- указать пользователя (идентификатор пользователя), под которым будет выполняться процесс в контейнере;
- не допустить контейнеру выполняться в качестве root (стандартный пользователь, в качестве которого контейнер работает, обычно определяется в самом образе контейнера, поэтому вам может потребоваться не допустить контейнерам работать в качестве root);
- запустить контейнер в привилегированном режиме, предоставив ему полный доступ к ядру узла;
- сконфигурировать тонко настроенные привилегии, добавляя или удаляя возможности – в отличие от предоставления контейнеру всех возможных разрешений, запуская его в привилегированном режиме;
- установить параметры SELinux (Linux с улучшенной безопасностью), для того чтобы накрепко запереть контейнер;
- не дать процессу записывать в файловую систему контейнера.

Мы рассмотрим эти варианты далее.

Выполнение модуля без указания контекста безопасности

Сначала запустите модуль со стандартными параметрами контекста безопасности (не указывая их вообще), чтобы можно было увидеть, как он ведет себя по сравнению с модулями с индивидуально настроенным контекстом безопасности:

```
$ kubectl run pod-with-defaults --image alpine --restart Never
➥   -- /bin/sleep 999999
pod "pod-with-defaults" created
```

Давайте посмотрим, каков идентификатор пользователя и группы, от имени которых контейнер работает, и к каким группам он принадлежит. Это можно увидеть, выполнив команду id внутри контейнера:

```
$ kubectl exec pod-with-defaults id
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon), 3(sys), 4(adm),
    6(disk), 10(wheel), 11(floppy), 20(dialout), 26(tape), 27(video)
```

Контейнер выполняется от идентификатора пользователя (uid) 0, то есть как root, и как идентификатор группы (gid) 0 (тоже root). Он также является членом нескольких других групп.

> **ПРИМЕЧАНИЕ.** В образе контейнера указывается, от имени какого пользователя работает контейнер. В файле Dockerfile это делается с помощью директивы USER. Если этот параметр опущен, то контейнер выполняется как `root`.

Теперь вы запустите модуль, в котором контейнер будет работать от имени другого пользователя.

13.2.1 Выполнение контейнера от имени конкретного пользователя

Чтобы запустить модуль с идентификатором пользователя, отличным от того, который впечатан в образ контейнера, необходимо задать свойство `securityContext.runAsUser` модуля. Контейнер будет запускаться от имени пользователя `guest`, идентификатор которого в образе контейнера `alpine` равен `405`. Это показано в следующем ниже листинге.

Листинг 13.6. Запуск контейнеров от имени конкретного пользователя: pod-as-user-guest.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-as-user-guest
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 405
```

Необходимо указать ИД, а не имя пользователя (id 405 соответствует гостевому пользователю guest)

Теперь, чтобы увидеть эффект свойства `runAsUser`, выполните команду `id` в этом новом модуле, как вы делали раньше:

```
$ kubectl exec pod-as-user-guest id
uid=405(guest) gid=100(users)
```

Согласно запросу, контейнер выполняется от имени гостевого пользователя `guest`.

13.2.2 Недопущение работы контейнера в качестве root

Что делать, если вам все равно, от имени какого пользователя контейнер работает, но вместе с тем вы хотите помешать ему запускаться в качестве root?

Представьте, что модуль развернут с образом контейнера, который был создан с помощью директивы `USER daemon` в файле Dockerfile, заставляющей контейнер работать от имени пользователя `daemon`. Что делать, если злоумышлен-

ник получает доступ к хранилищу образов и размещает другой образ под один и тот же тег? Образ злоумышленника сконфигурирован на запуск в качестве пользователя root. Когда Kubernetes назначит новый экземпляр модуля, агент Kubelet загрузит образ злоумышленника и выполнит любой код, который тот в него поместит.

Несмотря на то что контейнеры в основном от хост-системы изолированы, запуск их процессов в качестве root по-прежнему считается плохой практикой. Например, если при монтировании каталога хоста в контейнер процесс, работающий в контейнере, выполняется из-под root, то он имеет полный доступ к смонтированному каталогу, но если он работает не как root, то этого не будет.

Для того чтобы предотвратить описанный выше сценарий атаки, вы можете указать, что контейнер модуля должен выполняться от имени пользователя без полномочий root, как показано в следующем ниже листинге.

Листинг 13.7. Недопущение работы контейнеров из-под root: pod-run-as-non-root.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-run-as-non-root
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsNonRoot: true
```

Этот контейнер может работать только от имени пользователя без полномочий root

Если развернуть данный модуль, то он будет назначен, но ему не будет разрешено запускаться:

```
$ kubectl get po pod-run-as-non-root
NAME                 READY  STATUS
pod-run-as-non-root    0/1    container has runAsNonRoot and image will run as root
```

Теперь, если кто-то вмешается в ваши образы контейнеров, то он недалеко уйдет.

13.2.3 Выполнение модулей в привилегированном режиме

Иногда модули должны делать все, что может сделать узел, на котором они работают, например использовать защищенные системные устройства или другой функционал ядра, который недоступен обычным контейнерам.

Примером такого модуля является модуль `kube-proxy`, который должен изменять правила `iptables` узла, чтобы заставить службы работать так, как было

продемонстрировано в главе 11. Если вы выполните инструкции в приложении B и развернете кластер с помощью `kubeadm`, то увидите, что каждый узел кластера запускает модуль `kube-proxy`, и вы можете изучить его спецификацию YAML и разобраться во всем используемом им специальном функционале.

Чтобы получить полный доступ к ядру узла, контейнер модуля работает в привилегированном режиме. Это достигается путем присвоения значения `true` свойству `privileged` в свойстве `securityContext` контейнера. Вы создадите привилегированный модуль из YAML в следующем ниже листинге.

Листинг 13.8. Модуль с привилегированным контейнером: pod-privileged.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-privileged
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      privileged: true        <- Этот контейнер будет работать
                                 в привилегированном режиме
```

Продолжайте и разверните этот модуль, чтобы его можно было сравнить с обычным модулем, который вы запустили ранее.

Если вы знакомы с Linux, то вы, возможно, знаете, что он имеет специальный файловый каталог /dev, который содержит файлы устройств для всех устройств в системе. Это не обычные файлы на диске, а специальные файлы, используемые для связи с устройствами. Давайте посмотрим, какие устройства видны в контейнере без привилегий, который вы развернули ранее (модуль `pod-with-defaults`), выведя список файлов в его каталоге /dev. Это показано в следующем ниже листинге.

Листинг 13.9. Список доступных устройств в модуле без привилегий

```
$ kubectl exec -it pod-with-defaults ls /dev
core           null           stderr              urandom
fd             ptmx           stdin               zero
full           pts            stdout
fuse           random         termination-log
mqueue         shm            tty
```

В данном списке показаны все устройства. Список довольно короткий. Теперь сравните его со следующим ниже списком. В нем показаны файлы устройств, которые может видеть ваш привилегированный модуль.

Листинг 13.10. Список доступных устройств в привилегированном модуле

```
$ kubectl exec -it pod-privileged ls /dev
autofs           snd              tty46
bsg              sr0              tty47
btrfs-control    stderr           tty48
core             stdin            tty49
cpu              stdout           tty5
cpu_dma_latency  termination-log  tty50
fd               tty              tty51
full             tty0             tty52
fuse             tty1             tty53
hpet             tty10            tty54
hwrng            tty11            tty55
...              ...              ...
```

Я не включил весь список, потому что он слишком длинный для книги, но очевидно, что список устройств намного длиннее, чем раньше. Привилегированный контейнер видит практически все устройства узла. Это значит, что он может свободно использовать любое устройство.

Например, мне пришлось использовать привилегированный режим, как этот, когда мне потребовался модуль, работающий на Raspberry Pi, для управления подключенными к нему светодиодами.

13.2.4 Добавление отдельных функциональных возможностей ядра в контейнер

В предыдущем разделе вы увидели один из способов предоставления контейнеру неограниченной мощи. В старые времена традиционные реализации UNIX различали только привилегированные и непривилегированные процессы, но в течение многих лет Linux поддерживает гораздо более мелкозернистую систему разрешений через возможности ядра.

Вместо того чтобы делать контейнер привилегированным и предоставлять ему неограниченные разрешения, гораздо более безопасный метод (с точки зрения безопасности) – предоставлять ему доступ только к тем функциональным возможностям ядра, которые ему действительно нужны. Kubernetes позволяет добавлять эти функциональные возможности в каждый контейнер или удалить часть из них, что дает возможность тонко настраивать разрешения контейнера и ограничивать влияние потенциального вторжения злоумышленника.

Например, контейнеру обычно не разрешается изменять системное время (время аппаратных часов). Это можно подтвердить, установив время в вашем модуле pod-with-defaults:

```
$ kubectl exec -it pod-with-defaults -- date +%T -s "12:00:00"
date: can't set date: Operation not permitted
```

Если вы хотите разрешить контейнеру изменять системное время, то вы можете добавить (add) функциональную возможность CAP_SYS_TIME в список возможностей контейнера, как показано в следующем ниже листинге.

Листинг 13.11. Добавление функциональной возможности CAP_SYS_TIME: pod-add-settime-capability.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-add-settime-capability
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      capabilities:
        add:
        - SYS_TIME
```

Функциональные возможности добавляются или удаляются в рамках свойства securityContext

Вы добавляете функциональную возможность SYS_TIME

ПРИМЕЧАНИЕ. Функциональные возможности ядра Linux, как правило, обозначаются префиксом CAP_. Но при указании их в секции spec модуля данный префикс следует исключить.

Если выполнить ту же команду в контейнере этого нового модуля, то системное время успешно изменится:

```
$ kubectl exec -it pod-add-settime-capability -- date +%T -s "12:00:00"
12:00:00

$ kubectl exec -it pod-add-settime-capability -- date
Sun May 7 12:00:03 UTC 2017
```

ПРЕДУПРЕЖДЕНИЕ. Если вы попробуете сделать это самостоятельно, имейте в виду, что это может привести к неработоспособности рабочего узла. В Minikube, несмотря на то что системное время было автоматически сброшено демоном протокола сетевого времени Network Time Protocol (NTP), мне пришлось перезагрузить виртуальную машину, чтобы назначить новые модули.

Вы можете убедиться, что время узла было изменено, проверив время на узле модуля. В моем случае я использую Minikube, поэтому у меня есть только один узел, и я могу получить его время следующим образом:

```
$ minikube ssh date
Sun May 7 12:00:07 UTC 2017
```

Добавление функциональных возможностей таким способом является намного более оптимальным, чем предоставление контейнеру полных привилегий с помощью `privileged: true`. По общему признанию, это требует от вас знать и понимать, что делает каждая функциональная возможность.

> **СОВЕТ.** Вы найдете список функциональных возможностей ядра Linux на справочных страницах Linux.

13.2.5 Удаление функциональных возможностей из контейнера

Вы видели, как добавлять функциональные возможности, но вы также можете удалять функциональные возможности, которые в противном случае могут быть доступны для контейнера. Например, функциональные возможности, предоставляемые контейнеру по умолчанию, включают возможность `CAP_CHOWN`, которая позволяет процессам изменять владельца файлов в файловой системе.

Это можно увидеть, изменив владельца каталога /tmp в модуле `pod-with-defaults` на гостевого пользователя `guest`, например:

```
$ kubectl exec pod-with-defaults chown guest /tmp
$ kubectl exec pod-with-defaults -- ls -la / | grep tmp
drwxrwxrwt 2 guest root 6 May 25 15:18 tmp
```

Для того чтобы не дать контейнеру это делать, вам нужно удалить функциональную возможность, указав ее в списке в свойстве `securityContext.capabilities.drop`, как показано в следующем ниже листинге.

Листинг 13.12. Удаление функциональной возможности из контейнера: pod-drop-chown-capability.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-drop-chown-capability
spec:
  containers:
  - name: main
    image: alpine
    command: [«/bin/sleep", "999999"]
    securityContext:
      capabilities:
        drop:
          - CHOWN
```

Вы не позволяете этому контейнеру изменять владельца файлов

Убрав функциональную возможность CHOWN, вы не сможете изменить владельца каталога /tmp в этом модуле:

```
$ kubectl exec pod-drop-chown-capability chown guest /tmp
chown: /tmp: Operation not permitted
```

Вы почти закончили исследование параметров контекста безопасности контейнера. Давайте рассмотрим еще один.

13.2.6 Недопущение записи процессами в файловую систему контейнера

Вам может потребоваться запретить процессам, работающим в контейнере, записывать в файловую систему контейнера и разрешить записывать только в смонтированные тома. Вы хотели бы сделать это в основном по соображениям безопасности.

Представим, что вы запускаете PHP-приложение со скрытой уязвимостью, позволяющей злоумышленнику писать в файловую систему. PHP-файлы добавляются в образ контейнера во время сборки и раздаются из файловой системы контейнера. Вследствие этой уязвимости злоумышленник может видоизменить эти файлы и внедрить в них вредоносный код.

Такие типы атак могут быть предотвращены путем недопущения записи контейнером в свою файловую систему, где обычно хранится исполняемый код приложения. Это делается путем присвоения значения `true` свойству `securityContext.readOnlyRootFilesystem` контейнера, как показано в следующем ниже листинге.

Листинг 13.13. Контейнер с файловой системой только для чтения: pod-with-readonly-filesystem.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readonly-filesystem
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      readOnlyRootFilesystem: true
    volumeMounts:
    - name: my-volume
      mountPath: /volume
      readOnly: false
  volumes:
  - name: my-volume
    emptyDir:
```

В файловую систему этого контейнера выполнять запись не получится...

...но запись в /volume разрешена, потому что там смонтирован том

Когда вы развернете этот модуль, контейнер будет выполняться из-под root, который имеет разрешения на запись в каталог /directory, но попытка записи туда файла не удастся:

```
$ kubectl exec -it pod-with-readonly-filesystem touch /new-file
touch: /new-file: Read-only file system
```

С другой стороны, допускается запись в смонтированный том:

```
$ kubectl exec -it pod-with-readonly-filesystem touch /volume/newfile
$ kubectl exec -it pod-with-readonly-filesystem -- ls -la /volume/newfile
-rw-r--r-- 1 root root 0 May 7 19:11 /mountedVolume/newfile
```

Как показано в данном примере, когда вы делаете файловую систему контейнера доступной только для чтения, вам, вероятно, потребуется смонтировать том в каждом каталоге, в который приложение записывает (например, журналы, дисковые кеши и т. д.).

> **СОВЕТ.** В целях повышения безопасности при запуске модулей в рабочей среде присваивайте свойству контейнера `readOnlyRootFilesystem` значение `true`.

Настройка параметров контекста безопасности на уровне модуля

Во всех этих примерах задается контекст безопасности отдельного контейнера. Некоторые из этих параметров также могут быть установлены на уровне модуля (посредством свойства `pod.spec.securityContext`). Они служат параметрами, действующими по умолчанию, для всех контейнеров модуля, но могут быть переопределены на уровне контейнера. Контекст безопасности уровня модуля также позволяет задавать дополнительные свойства, которые мы объясним далее.

13.2.7 Совместное использование томов, когда контейнеры запущены под разными пользователями

В главе 6 мы объяснили, как для обмена данными между контейнерами модуля используются тома. У вас не было проблем с записью файлов в одном контейнере и чтением их в другом.

Но это было только потому, что оба контейнера работали как root, что предоставляло им полный доступ ко всем файлам в томе. Теперь предположим, что вы используете параметр `runAsUser`, который мы объяснили ранее. Возможно, вам придется запускать два контейнера от имени двух разных пользователей (вероятно, вы воспользуетесь двумя сторонними образами контейнеров, где каждый запускает свой процесс от имени своего конкретного пользователя). Если эти два контейнера используют том для совместного использования файлов, они вовсе не обязательно смогут читать или записывать файлы друг друга.

Вот почему Kubernetes позволяет для всех модулей указывать дополнительные группы, работающие в контейнере, позволяя им обмениваться файлами независимо от идентификаторов пользователей, с которыми они работают. Это делается с помощью следующих двух свойств:

- fsGroup;
- supplementalGroups.

То, что они делают, лучше всего объяснить на примере, поэтому давайте посмотрим, как их использовать в модуле, а затем посмотрим, каков их эффект. В следующем ниже листинге описывается модуль с двумя контейнерами, совместно использующими один и тот же том.

Листинг 13.14. Группы fsGroup и supplementalGroups: pod-with-shared-volume-fsgroup.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-shared-volume-fsgroup
spec:
  securityContext:
    fsGroup: 555
    supplementalGroups: [666, 777]
  containers:
  - name: first
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 1111
   volumeMounts:
    - name: shared-volume
      mountPath: /volume
      readOnly: false
  - name: second
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 2222
    volumeMounts:
    - name: shared-volume
      mountPath: /volume
      readOnly: false
  volumes:
  - name: shared-volume
    emptyDir:
```

После создания этого модуля запустите оболочку в его первом контейнере и посмотрите, какие идентификаторы пользователей и групп работают в этом контейнере:

```
$ kubectl exec -it pod-with-shared-volume-fsgroup -c first sh
/ $ id
uid=1111 gid=0(root) groups=555,666,777
```

Команда `id` показывает, что данный контейнер выполняется с идентификатором пользователя `1111`, как указано в определении модуля. Фактический идентификатор группы равен `0` (`root`), но идентификаторы группы `555`, `666` и `777` также связаны с пользователем.

В определении модуля вы присвоили `fsGroup` значение `555`. По этой причине, как показано ниже, смонтированный том будет принадлежать идентификатору группы `555`:

```
/ $ ls -l / | grep volume
drwxrwsrwx    2 root     555             6 May 29 12:23 volume
```

Если вы создаете файл в каталоге смонтированного тома, то этот файл принадлежит пользователю с идентификатором `1111` (это идентификатор пользователя, под которым выполняется контейнер) и группе с идентификатором `555`:

```
/ $ echo foo > /volume/foo
/ $ ls -l /volume
total 4
-rw-r--r--     1 1111     555             4 May 29 12:25 foo
```

Это отличается от того, как для вновь создаваемых файлов настраивается владение в иных случаях. Обычно, когда пользователь создает файлы, используется фактический идентификатор группы пользователя, который в вашем случае равен `0`. Это можно увидеть, создав файл не в томе, а в файловой системе контейнера:

```
/ $ echo foo > /tmp/foo
/ $ ls -l /tmp
total 4
-rw-r--r--     1 1111     root            4 May 29 12:41 foo
```

Как вы можете видеть, свойство контекста безопасности `fsGroup` используется, когда процесс создает файлы в томе (но это зависит от используемого плагина тома), тогда как свойство `supplementalGroups` определяет список дополнительных идентификаторов групп, с которыми связан пользователь.

На этом раздел о конфигурации контекста безопасности контейнера завершается. Далее мы рассмотрим то, как администратор кластера может ограничивать пользователей в этой работе.

13.3 Ограничение использования функциональности, связанной с безопасностью в модулях

Примеры в предыдущих разделах показали, как человек, развертывающий модули, может делать все, что угодно, на любом узле кластера, к примеру развертывая привилегированный модуль на узле. Очевидно, что должен иметься механизм, который не дает пользователям делать часть или все, что было объяснено. Администратор кластера может ограничивать использование ранее описанных функциональных средств обеспечения безопасности, создав один или несколько ресурсов политики безопасности модуля `PodSecurityPolicy`.

13.3.1 Знакомство с ресурсами PodSecurityPolicy

Политика безопасности модуля PodSecurityPolicy – это ресурс кластерного уровня (без пространства имен), который определяет, какие функциональные средства безопасности пользователи могут или не могут использовать в своих модулях. Задача поддержки политик, сконфигурированных в ресурсах политики безопасности PodSecurityPolicy, выполняется плагином управления допуском PodSecurityPolicy, работающим на сервере API (мы дали пояснения относительно плагинов управления допуском в главе 11).

> **ПРИМЕЧАНИЕ.** Плагин управления допуском PodSecurityPolicy может оказаться не активированным в кластере. Перед выполнением следующих далее примеров убедитесь, что он активирован. Если вы используете Minikube, обратитесь к следующей ниже вставке.

Когда кто-то отправляет ресурс модуля на сервер API, плагин управления допуском PodSecurityPolicy проверяет определение модуля относительно сконфигурированных политик PodSecurityPolicy. Если модуль соответствует политикам кластера, то он принимается и сохраняется в хранилище etcd; в противном случае он немедленно отклоняется. Данный плагин может также модифицировать ресурс модуля в соответствии со значениями, действующими по умолчанию, заданными в политике.

Что может делать политика безопасности модуля

Ресурс политики безопасности модуля PodSecurityPolicy определяет следующее:

- может ли модуль использовать пространства имен хоста IPC, PID или Network;
- к каким портам хоста может быть привязан модуль;
- под какими идентификаторами пользователя может работать контейнер;

- можно ли создать модуль с привилегированными контейнерами;
- какие функциональные возможности ядра разрешены, какие добавляются по умолчанию и какие всегда игнорируются;
- какие метки SELinux может использовать контейнер;
- может ли контейнер использовать доступную для записи корневую файловую систему;
- под какими группами файловых систем может работать контейнер;
- какие типы томов может использовать модуль.

Если вы прочитали данную главу до этого места, то все пункты, кроме последнего, в приведенном выше списке должны быть знакомы. Последний пункт также должен быть достаточно ясным.

Активирование управления ролевым доступом RBAC и плагина управления допуском PodSecurityPolicy в Minikube

Для выполнения этих примеров я использую Minikube версии v0.19.0. Эта версия не активирует плагин управления допуском PodSecurityPolicy или авторизацию RBAC, которая требуется в части упражнений. Одно из упражнений также требует аутентификации в качестве другого пользователя, поэтому вам нужно активировать и плагин обычной аутентификации, где пользователи определяются в файле.

Для запуска Minikube, в котором все эти плагины активированы, вы можете применить приведенную ниже команду (или аналогичную), в зависимости от версии, которую вы используете:

```
$ minikube start --extra-config apiserver.Authentication.PasswordFile.
➥ BasicAuthFile=/etc/kubernetes/passwd --extra-config=apiserver.
➥ Authorization.Mode=RBAC --extra-config=apiserver.GenericServerRun
➥ Options.AdmissionControl=NamespaceLifecycle,LimitRanger,Service
➥ Account,PersistentVolumeLabel,DefaultStorageClass,ResourceQuota,
➥ DefaultTolerationSeconds,PodSecurityPolicy
```

Сервер API не будет запускаться до тех пор, пока не будет создан файл паролей, указанный в параметрах командной строки. Ниже показано, как этот файл создается:

```
$ cat <<EOF | minikube ssh sudo tee /etc/kubernetes/passwd
password,alice,1000,basic-user
password,bob,2000,privileged-user
EOF
```

Вы найдете шелл-скрипт, который выполняет обе команды, в архиве кода этой книги в `Chapter13/minikube-with-rbac-and-psp-enabled.sh`.

Исследование примера политики безопасности модуля

В следующем ниже листинге показан пример политики PodSecurityPolicy, который запрещает модулям использовать пространства имен IPC, PID и Network узла, а также запрещает запуск привилегированных контейнеров и использование большинства портов узла (кроме портов 10 000–11 000 и 13 000–14 000). Политика не устанавливает ограничений для пользователей, групп или групп SELinux, от имени которых может выполняться контейнер.

Листинг 13.15. Пример политики безопасности модуля: pod-security-policy.yaml

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: default
spec:
  hostIPC: false
  hostPID: false
  hostNetwork: false
  hostPorts:
  - min: 10000
    max: 11000
  - min: 13000
    max: 14000
  privileged: false
  readOnlyRootFilesystem: true
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  volumes:
  - '*'
```

Контейнерам не разрешается использовать пространства имен хоста IPC, PID или Network

Они могут привязываться только к портам хоста от 10 000 до 11 000 (включительно) или портам хоста от 13 000 до 14 000

Контейнеры не могут работать в привилегированном режиме

Контейнеры принудительно запускаются с корневой файловой системой, доступной только для чтения

Контейнеры могут работать от имени любого пользователя и любой группы

Они также могут использовать любые группы SELinux, которые они хотят

В модулях можно использовать все типы томов

Большинство параметров, указанных в этом примере, должны быть понятными, в особенности если вы прочитали предыдущие разделы. После того как этот ресурс PodSecurityPolicy отправлен в кластер, сервер API больше не разрешит вам разворачивать использованный ранее привилегированный модуль. Например:

```
$ kubectl create -f pod-privileged.yaml
Error from server (Forbidden): error when creating "pod-privileged.yaml":
pods "pod-privileged" is forbidden: unable to validate against any pod
security policy: [spec.containers[0].securityContext.privileged: Invalid
value: true: Privileged containers are not allowed]
```

Точно так же вы больше не сможете развертывать модули, которые хотят использовать пространства имен хоста PID, IPC или Network. Кроме того, поскольку вы назначили значение `true` параметру `readOnlyRootFilesystem` в политике, файловые системы контейнеров во всех модулях будут доступны только для чтения (контейнеры могут выполнять запись только в тома).

13.3.2 Политики runAsUser, fsGroup и supplementalGroups

Политика в предыдущем примере не накладывает никаких лимитов на запуск контейнеров пользователей и групп, поскольку для полей `runAsUser`, `fsGroup` и `supplementalGroups` вы использовали правило `RunAsAny`. Если вы хотите ограничить список разрешенных идентификаторов пользователей или групп, измените правило на `MustRunAs` и укажите диапазон разрешенных идентификаторов.

Использование правила MustRunAs

Давайте рассмотрим пример. Для того чтобы разрешить контейнерам работать только с идентификатором пользователя `2` и ограничить идентификаторы группы файловой системы по умолчанию и дополнительной группы диапазонами `2–10` или `20–30` (оба включительно), необходимо включить в ресурс PodSecurityPolicy следующий фрагмент кода.

Листинг 13.16. Указание идентификаторов, под которыми должны работать контейнеры: psp-must-run-as.yaml

```
runAsUser:
  rule: MustRunAs
  ranges:
  - min: 2
    max: 2
fsGroup:
  rule: MustRunAs
  ranges:
  - min: 2
    max: 10
  - min: 20
    max: 30
supplementalGroups:
  rule: MustRunAs
  ranges:
  - min: 2
    max: 10
  - min: 20
    max: 30
```

Добавить единственный диапазон, в котором min равен max, чтобы установить один конкретный идентификатор

Поддерживается несколько диапазонов – здесь идентификаторы групп могут быть 2–10 или 20–30 (включительно)

Если секция `spec` модуля пытается назначить любому из этих полей значение за пределами указанных диапазонов, то модуль не будет принят сер-

вером API. Для того чтобы это попробовать, удалите предыдущую политику PodSecurityPolicy и создайте новую из файла psp-must-run-as.yaml.

> **ПРИМЕЧАНИЕ.** Смена политики не повлияет на существующие модули, потому что политика PodSecurityPolicy применяется только при создании или обновлении модулей.

Развертывание модуля со свойством RunAsUser вне диапазона политики

Если вы попробуете развернуть ранее указанный файл pod-as-user-guest.yaml, который говорит, что контейнер должен работать под идентификатором пользователя `405`, то сервер API этот модуль отклонит:

```
$ kubectl create -f pod-as-user-guest.yaml
Error from server (Forbidden): error when creating "pod-as-user-guest.yaml"
: pods "pod-as-user-guest" is forbidden: unable to validate against any pod
security policy: [securityContext.runAsUser: Invalid value: 405: UID on
container main does not match required range. Found 405, allowed: [{2 2}]]
```

Ладно, это было очевидно. Но что произойдет, если развернуть модуль без установки свойства `runAsUser` и если идентификатор пользователя впечатан в образ контейнера (с помощью директивы `USER` в файле Dockerfile)?

Развертывание модуля с образом контейнера с идентификатором пользователя вне диапазона

Для приложения Node.js, которое вы использовали на протяжении всей книги, я создал альтернативный образ. Указанный образ настроен таким образом, что контейнер будет работать под идентификатором пользователя `5`. Файл Dockerfile для данного образа показан в следующем ниже листинге.

Листинг 13.17. Файл Dockerfile с директивой USER: kubia-run-as-user-5/Dockerfile

```
FROM node:7
ADD app.js /app.js
USER 5                          <- Контейнеры, запускаемые из этого образа, будут
                                   работать под идентификатором пользователя 5
ENTRYPOINT ["node", "app.js"]
```

Я отправил этот образ в хранилище Docker Hub как `luksa/kubia-run-as-user-5`. Если я разверну модуль с помощью данного образа, то сервер API его не отклонит:

```
$ kubectl run run-as-5 --image luksa/kubia-run-as-user-5 --restart Never
pod "run-as-5" created
```

В отличие от того, что было ранее, сервер API принял модуль, и агент Kubelet запустил его контейнер. Давайте посмотрим, под каким идентификатором пользователя контейнер работает:

```
$ kubectl exec run-as-5 -- id
uid=2(bin) gid=2(bin) groups=2(bin)
```

Как вы можете видеть, контейнер работает под идентификатором пользователя 2, то есть идентификатором, указанным в политике PodSecurityPolicy. Политика PodSecurityPolicy может использоваться для переопределения идентификатора пользователя, жестко закодированного в образе контейнера.

Использование правила MustRunAsNonRoot в поле RunAsUser

Для поля `runAsUser` можно использовать дополнительное правило: `MustRunAsNonRoot`. Как следует из названия, оно запрещает пользователям развертывать контейнеры, работающие от имени root. Либо секция `spec` контейнера должна указывать поле `runAsUser`, которое не может быть равно нулю (ноль – это идентификатор пользователя `root`), либо сам образ контейнера должен выполняться под ненулевым идентификатором пользователя. Ранее мы объяснили, почему это хорошо.

13.3.3 Конфигурирование разрешенных, стандартных и запрещенных возможностей

Как вы узнали, контейнеры могут работать в привилегированном режиме или без него, и вы можете определять тонко настроенную конфигурацию разрешений, добавляя или удаляя в каждом контейнере функциональные возможности ядра Linux. Три поля влияют на то, какие функциональные возможности могут или не могут использовать контейнеры:

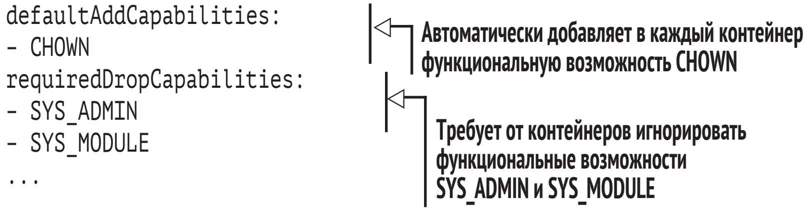
- `allowedCapabilities`;
- `defaultAddCapabilities`;
- `requiredDropCapabilities`.

Сначала мы рассмотрим пример, а затем обсудим, что делает каждое из трех полей. Ниже приведен фрагмент ресурса PodSecurityPolicy, определяющего три поля, связанных с функциональными возможностями.

Листинг 13.18. Указание функциональных возможностей в политике PodSecurityPolicy: psp-capabilities.yaml

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
spec:
  allowedCapabilities:
  - SYS_TIME
```

Позволяет контейнерам добавлять функциональную возможность SYS_TIME



ПРИМЕЧАНИЕ. Функциональная возможность SYS_ADMIN позволяет выполнять целый ряд административных операций, и функциональная возможность SYS_MODULE позволяет загружать и выгружать модули ядра Linux.

Указание функциональных возможностей, которые можно добавлять в контейнер

Поле `allowedCapabilities` используется для указания, какие функциональные возможности авторы модулей могут добавлять в поле `securityContext.capabilities` в секции `spec` контейнера. В одном из предыдущих примеров в контейнер была добавлена функциональная возможность `SYS_TIME`. Если бы плагин управления допуском PodSecurityPolicy был активирован, то вы не смогли бы добавить эту функциональную возможность, если только она не была указана в политике PodSecurityPolicy, как показано в листинге 13.18.

Добавление функциональных возможностей во все контейнеры

Все функциональные возможности, перечисленные в поле `defaultAddCapabilities`, будут добавлены в контейнеры каждого развернутого модуля. Если пользователь не хочет, чтобы определенные контейнеры имели эти функциональные возможности, он должен явным образом их удалить из секции `spec` этих контейнеров.

Пример в листинге 13.18 позволяет автоматически добавлять в каждый контейнер функциональную возможность `CAP_CHOWN`, которая позволяет процессам, работающим в контейнере, менять владельца файлов в контейнере (например, с помощью команды `chown`).

Удаление функциональных возможностей из контейнера

Последнее поле в этом примере – `requiredDropCapabilities`. Должен признать, что для меня поначалу это имя показалось несколько странным, но все оказалось не так сложно. Функциональные возможности, перечисленные в этом поле, удаляются автоматически из каждого контейнера (плагин управления допуском PodSecurityPolicy добавит их в поле `securityContext.capabilities.drop` каждого контейнера).

Если пользователь попытается создать модуль, где он явным образом добавляет одну из функциональных возможностей, перечисленных в поле `requiredDropCapabilities` политики, то модуль отклоняется:

```
$ kubectl create -f pod-add-sysadmin-capability.yaml
Error from server (Forbidden): error when creating "pod-add-sysadmincapability.yaml":
pods "pod-add-sysadmin-capability" is forbidden: unable
to validate against any pod security policy: [capabilities.add: Invalid
value: "SYS_ADMIN": capability may not be added]
```

13.3.4 Ограничение типов томов, которые модули могут использовать

Последнее, что может делать ресурс PodSecurityPolicy, – это определять, какие типы томов пользователи могут добавлять в свои модули. Как минимум, политика PodSecurityPolicy должна позволять использовать, по крайней мере, тома `emptyDir`, `configMap`, `secret`, `downwardAPI` и `persistentVolumeClaim`. Соответствующая часть такого ресурса PodSecurityPolicy показана в следующем ниже листинге.

Листинг 13.19. Фрагмент ресурса PodSecurityPolicy, позволяющий использовать только определенные типы томов: psp-volumes.yaml

```
kind: PodSecurityPolicy
spec:
  volumes:
  - emptyDir
  - configMap
  - secret
  - downwardAPI
  - persistentVolumeClaim
```

Если имеется несколько ресурсов PodSecurityPolicy, то модули могут использовать любой тип тома, определенный в любой политике (используется объединение всех списков томов).

13.3.5 Назначение разных политик PodSecurityPolicy разным пользователям и группам

Мы упомянули, что политика безопасности модуля PodSecurityPolicy является ресурсом кластерного уровня, что означает, что она не может быть сохранена и применена к определенному пространству имен. Означает ли это, что это всегда применимо во всех пространствах имен? Нет, потому что это сделало бы их относительно непригодными для использования. В конце концов, системным модулям часто нужно позволять делать вещи, которые регулярные модули не должны делать.

Назначение разных политик разным пользователям выполняется с помощью механизма RBAC, описанного в предыдущей главе. Идея состоит в том, чтобы создавать столько политик, сколько вам нужно, и делать их доступными для отдельных пользователей или групп, создавая ресурсы кластерной

роли ClusterRole и указывая их на отдельные политики по имени. Привязывая эти кластерные роли к конкретным пользователям или группам с помощью привязок ClusterRoleBinding, когда плагин управления допуском политики PodSecurityPolicy должен решать, следует допускать определение модуля или нет, он будет рассматривать только те политики, которые доступны пользователю, создающему модуль.

Вы увидите, как это сделать, в следующем упражнении. Но сейчас вы начнете с создания дополнительной политики PodSecurityPolicy.

Создание политики безопасности модуля, позволяющей развертывать привилегированные контейнеры

Вы создадите специальную политику PodSecurityPolicy, которая позволит привилегированным пользователям создавать модули с привилегированными контейнерами. В следующем ниже листинге показано определение политики.

Листинг 13.20. Политика PodSecurityPolicy для привилегированных пользователей: psp-privileged.yaml

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged        <-- Имя этой политики – «привилегированная»
spec:
  privileged: true        <-- Она позволяет запускать привилегированные контейнеры
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  volumes:
  - '*'
```

После того как вы отправите эту политику на сервер API, в кластере будет иметься две политики:

```
$ kubectl get psp
NAME         PRIV    CAPS   SELINUX    RUNASUSER   FSGROUP    ...
default      false   []     RunAsAny   RunAsAny    RunAsAny   ...
privileged   true    []     RunAsAny   RunAsAny    RunAsAny   ...
```

ПРИМЕЧАНИЕ. Аббревиатурой для `PodSecurityPolicy` является `psp`.

Как вы можете видеть в столбце `PRIV`, политика `default` не позволяет запускать привилегированные контейнеры, тогда как привилегированная политика это делает. Поскольку вы вошли в систему как администратор кластера, то можете увидеть все политики. Если при создании модулей какая-либо политика позволяет развернуть модуль с определенными функциональными особенностями, сервер API примет этот модуль.

Теперь представьте, что ваш кластер используют два дополнительных пользователя: Алиса и Боб. Вы хотите, чтобы Алиса развертывала только ограниченные (непривилегированные) модули, при этом вы хотите разрешить Бобу также развертывать привилегированные модули. Вы делаете это, убедившись, что Алиса может использовать лишь политику PodSecurityPolicy `default`, а Бобу разрешено использовать обе.

Использование RBAC для назначения разных политик безопасности модуля разным пользователям

В предыдущей главе вы использовали управление ролевым доступом RBAC для предоставления пользователям доступа только к определенным типам ресурсов, но я упомянул, что доступ может быть предоставлен определенным экземплярам ресурсов путем ссылки на них по имени. Это то, что вы будете использовать, чтобы дать пользователям возможность применять разные ресурсы PodSecurityPolicy.

Прежде всего вы создадите две кластерные роли ClusterRole, причем каждая позволяет использовать одну из политик. Вы назовете первую `psp-default` и в ней разрешите использование ресурса PodSecurityPolicy `default`. Для этого вы можете применить команду `kubectl create clusterrole`:

```
$ kubectl create clusterrole psp-default --verb=use
➥   --resource=podsecuritypolicies --resource-name=default
clusterrole "psp-default" created
```

> **ПРИМЕЧАНИЕ.** Вместо `get`, `list`, `watch` или похожих глаголов вы используете специальный глагол `use`.

Как вы можете видеть, вы ссылаетесь на конкретный экземпляр ресурса PodSecurityPolicy, используя параметр `--resource-name`. Теперь создайте другую кластерную роль ClusterRole под названием `psp-privileged`, указывающую на привилегированную политику:

```
$ kubectl create clusterrole psp-privileged --verb=use
➥   --resource=podsecuritypolicies --resource-name=privileged
clusterrole "psp-privileged" created
```

Теперь вам нужно привязать эти две политики к пользователям. Как вы, возможно, помните из предыдущей главы, если вы привязываете кластерную роль, которая предоставляет доступ к ресурсам уровня кластера (которыми

являются ресурсы PodSecurityPolicy), то вместо привязки (с пространством имен) RoleBinding вам нужно использовать привязку ClusterRoleBinding.

Вы собираетесь привязать кластерную роль `psp-default` ко всем аутентифицированным пользователям, а не только к Алисе. Эта необходимость вызвана тем, что в противном случае никто не сможет создавать какие-либо модули, поскольку плагин контроля допуска будет жаловаться, что никакой политики не существует. Все аутентифицированные пользователи входят в состав группы `system:authenticated`, поэтому вы привяжете указанную кластерную роль к этой группе:

```
$ kubectl create clusterrolebinding psp-all-users
➥   --clusterrole=psp-default --group=system:authenticated
clusterrolebinding "psp-all-users" created
```

Вы привяжете кластерную роль `psp-privileged` только к Бобу:

```
$ kubectl create clusterrolebinding psp-bob
➥   --clusterrole=psp-privileged --user=bob
clusterrolebinding "psp-bob" created
```

Как аутентифицированный пользователь Алиса должна теперь иметь доступ к политике безопасности `default`, тогда как Боб должен иметь доступ к обеим политикам: и `default`, и `privileged`. Алиса не должна иметь возможность создавать привилегированные модули, в то время как Боб должен. Посмотрим, так ли это.

Создание дополнительных пользователей для kubectl

Но как вы аутентифицируетесь как Алиса или Боб независимо от того, в качестве кого вы аутентифицированы в настоящее время? В приложении A данной книги объясняется, как `kubectl` может использоваться с несколькими кластерами, а также несколькими контекстами. Контекст включает учетные данные пользователя, используемые для обмена с кластером. Обратитесь к приложению A, чтобы узнать подробности. Здесь же мы покажем голые команды, позволяющие использовать `kubectl` в качестве Алисы или Боба.

Сначала вы создадите двух новых пользователей в конфигурации `kubectl` с помощью следующих двух команд:

```
$ kubectl config set-credentials alice --username=alice --password=password
User "alice" set.
$ kubectl config set-credentials bob --username=bob --password=password
User "bob" set.
```

То, что эти команды делают, должно быть совершенно очевидно. Поскольку вы задаете учетные данные имени пользователя и пароля, `kubectl` будет использовать для этих двух пользователей обычную аутентификацию HTTP (другие методы аутентификации включают токены, клиентские сертификаты и т. д.).

Создание модулей от имени другого пользователя

Теперь вы можете попробовать создать привилегированный модуль, аутентифицировавшись как Алиса. С помощью параметра `--user` вы можете сообщить `kubectl`, учетные данные какого пользователя применять:

```
$ kubectl --user alice create -f pod-privileged.yaml
Error from server (Forbidden): error when creating "pod-privileged.yaml":
    pods "pod-privileged" is forbidden: unable to validate against any pod
    security policy: [spec.containers[0].securityContext.privileged: Invalid
    value: true: Privileged containers are not allowed]
```

Как и ожидалось, сервер API не позволяет Алисе создавать привилегированные модули. Теперь давайте посмотрим, позволяет ли он это Бобу:

```
$ kubectl --user bob create -f pod-privileged.yaml
pod "pod-privileged" created
```

И вот, пожалуйста. Вы успешно применили управление ролевым доступом RBAC, чтобы заставить плагин управления допуском использовать разные ресурсы PodSecurityPolicy для разных пользователей.

13.4 Изоляция сети модулей

До сих пор в этой главе мы исследовали множество связанных с безопасностью параметров конфигурации, которые применяются в отдельных модулях и их контейнерах. В оставшейся части данной главы мы рассмотрим, каким образом сеть между модулями может быть защищена путем ограничения того, как одни модули взаимодействуют с другими модулями.

Ответ на вопрос, можно это сконфигурировать или нет, зависит от того, какой в кластере используется плагин контейнерного сетевого взаимодействия. Если плагин сетевого взаимодействия его поддерживает, то вы можете сконфигурировать сетевую изоляцию, создав ресурсы политики сети NetworkPolicy.

Политика сети NetworkPolicy применяется к модулям, которые совпадают с ее селектором меток, и указывает либо на то, какие источники могут получать доступ к совпавшим модулям, либо на то, к каким целевым назначениям можно получить доступ из совпавших модулей. Это настраивается посредством правил соответственно входа (ingress) и выхода (egress). Оба типа правил могут отождествлять только те модули, которые соответствуют селектору модулей, все модули в пространстве имен, чьи метки совпадают с селектором пространства имен, либо сетевым IP-блоком, указанным с помощью обозначения бесклассовой междоменной маршрутизации (Classless Inter-Domain Routing, CIDR) (например, 192.168.1.0/24).

Мы рассмотрим правила входа и выхода и все три варианта соответствия.

ПРИМЕЧАНИЕ. Правила входа ingress в политике сети NetworkPolicy не имеют ничего общего с ресурсом Ingress, рассмотренным в главе 5.

13.4.1 Активация изоляции сети в пространстве имен

По умолчанию доступ к модулям в заданном пространстве имен может получить любой пользователь. Прежде всего вам нужно это изменить. Вы создадите политику сети `default-deny`, которая не позволяет всем клиентам подключаться к любому модулю в вашем пространстве имен. Определение политики сети NetworkPolicy показано в следующем ниже листинге.

Листинг 13.21. Политика сети default-deny: network-policy-default-deny.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
```

Пустой селектор модулей соответствует всем модулям в том же пространстве имен

Когда вы создаете эту политику сети в определенном пространстве имен, никто не имеет возможности подключаться к любым модулям в этом пространстве имен.

> **ПРИМЕЧАНИЕ.** Плагин контейнерного сетевого взаимодействия CNI или другой тип решения сетевого взаимодействия, используемого в кластере, должен поддерживать политику сети, иначе не будет никакого эффекта на межмодульную связность.

13.4.2 Разрешение подключения к серверному модулю только некоторых модулей в пространстве имен

Для того чтобы разрешить клиентам подключаться к модулям в пространстве имен, необходимо явно указать, кто именно может подключаться к модулям. Под «кто» подразумеваются, какие именно модули. Давайте рассмотрим, как это сделать, на примере.

Представьте, что имеется модуль базы данных PostgreSQL, работающий в пространстве имен `foo`, и веб-серверный модуль, который использует базу данных. В этом пространстве имен есть и другие модули, и вы не хотите разрешать им подключаться к базе данных. Для защиты сети необходимо создать ресурс NetworkPolicy, показанный в следующем ниже листинге, в том же пространстве имен, что и модуль базы данных.

Листинг 13.22. Политика сети для модуля PostgreSQL: network-policy-postgres.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: postgres-netpolicy
spec:
  podSelector:
    matchLabels:
      app: database
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: webserver
    ports:
    - port: 5432
```

Эта политика обеспечивает доступ к модулям с меткой app=database

Разрешает входящие соединения только от модулей с меткой app=webserver

Подключения к этому порту разрешены

Данный пример политики сети позволяет модулям с меткой `app=webserver` подключаться к модулям с меткой `app=database` и только на порту 5432. Другие модули не могут подключаться к модулям базы данных, и никто (даже веб-серверные модули) не может подключаться ни к чему, кроме порта 5432 модулей базы данных. Это показано на рис. 13.4.

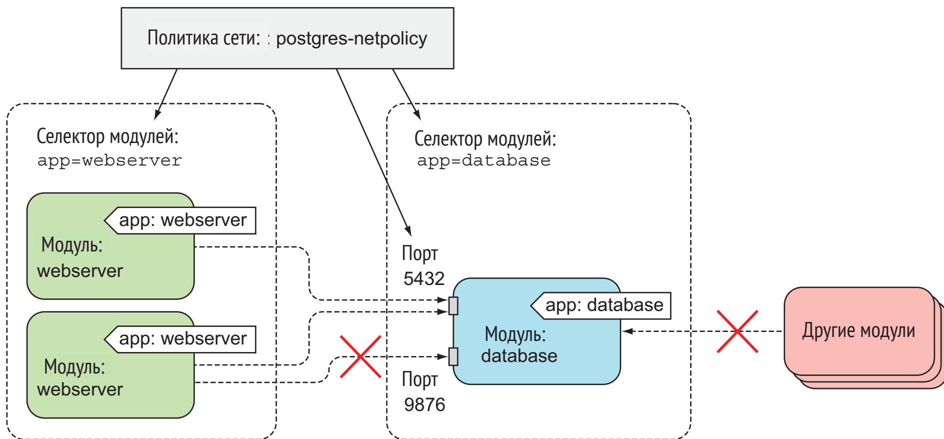


Рис. 13.4. Политика сети, разрешающая только некоторым модулям обращаться к другим модулям и только на определенном порту

Клиентские модули обычно подключаются к серверным модулям через службу, а не непосредственно к модулю, но это ничего не меняет. Политика сети NetworkPolicy также обеспечивается при подключении через службу.

13.4.3 Изоляция сети между пространствами имен Kubernetes

Теперь давайте рассмотрим еще один пример, где несколько пользователей используют один и тот же кластер Kubernetes. Каждый пользователь может использовать несколько пространств имен, и каждое пространство имен имеет метку, конкретизирующую пользователя, которому оно принадлежит. Напри-

мер, один из этих пользователей – издательство Manning. Все его пространства имен помечены как `tenant: manning`. В одном из своих пространств имен оно запускает микросервис корзины покупок, которая должна быть доступна для всех модулей, работающих в любом из его пространств имен. Очевидно, оно не хочет, чтобы какие-либо другие пользователи имели доступ к его микросервису.

Чтобы обезопасить свой микросервис, оно создает ресурс политики сети NetworkPolicy, показанной в следующем ниже листинге.

Листинг 13.23. Политика сети для модуля (модулей) корзины покупок: network-policy-cart.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: shoppingcart-netpolicy
spec:
  podSelector:
    matchLabels:
      app: shopping-cart
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          tenant: manning
    ports:
    - port: 80
```

Эта политика применяется к модулям, помеченным как microservice=shopping-cart

Доступ к микросервису разрешен только тем модулям, которые работают в пространствах имен, помеченных как tenant=manning

Как показано на рис. 13.5, эта политика сети гарантирует, что доступ к его микросервису корзины покупок могут получать только те модули, которые работают в пространствах имен, помеченных как `tenant: manning`.

Если провайдер корзины также хочет предоставить доступ другим пользователям (возможно, одной из компаний-партнеров), то он может либо создать дополнительный ресурс политики сети NetworkPolicy, либо добавить дополнительное правило входа в существующую политику сети NetworkPolicy.

ПРИМЕЧАНИЕ. В кластере Kubernetes со множеством пользователей пользователи обычно не могут добавлять в свои пространства имен метки (либо аннотации). Если бы они могли, то они могли бы обойти правила входа, основанные на селекторе `namespaceSelector`.

13.4.4 Изоляция с использованием обозначения CIDR

Вместо указания селектора модулей или пространств имен для определения того, кто может обращаться к модулям, намеченным в политике сети, вы также можете указать IP-блок в обозначении CIDR. Например, для того чтобы

разрешить доступ к модулям shopping-cart из предыдущего раздела только из IP-адресов в диапазоне от 192.168.1.1 до 255, вы укажете правило входа в следующем ниже листинге.

Листинг 13.24. Указание IP-блока в правиле входа: network-policy-cidr.yaml

```
ingress:
- from:
  - ipBlock:
      cidr: 192.168.1.0/24
```

Это правило входа допускает трафик только от клиентов в IP-блоке 192.168.1.0/24

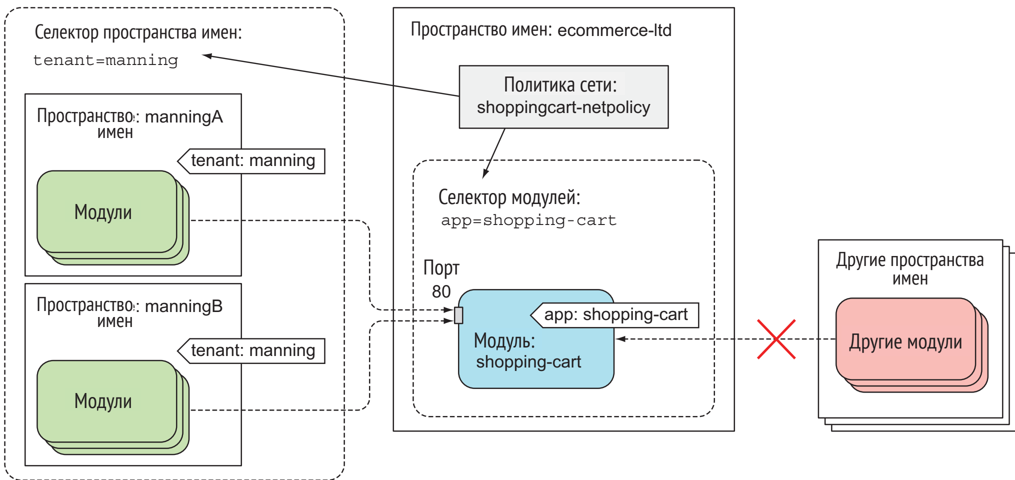


Рис. 13.5. Политика сети, разрешающая доступ к конкретному модулю только модулям в пространствах имен, совпадающих с селектором namespaceSelector

13.4.5 Лимитирование исходящего трафика набора модулей

Во всех предыдущих примерах входящий трафик лимитировался модулями, которые совпадают с селектором модулей политики сети с помощью правил входа. Помимо этого, вы также можете лимитировать их исходящий трафик с помощью правил выхода. Соответствующий пример показан в следующем ниже листинге.

Листинг 13.25. Использование правил выхода в политике сети: network-policy-egress.yaml

```
spec:
  podSelector:
    matchLabels:
      app: webserver
  egress:
```

Эта политика применяется к модулям с меткой app=webserver

Лимитирует исходящий трафик модулей

```
- to:
  - podSelector:
      matchLabels:
        app: database
```

Политика сети в предыдущем листинге разрешает модулям, которые имеют метку `app=webserver`, иметь доступ только к тем модулям, которые имеют метку `app=database`, и ни к чему более (ни к другим модулям, ни к каким-либо другим IP-адресам, независимо от того, является ли он внутренним или внешним по отношению к кластеру).

13.5 Резюме

В этой главе вы познакомились с защитой узлов кластера от модулей и модулей от других модулей. Вы узнали, что:

- модули могут использовать пространства имен Linux узла вместо собственных;
- контейнеры могут быть сконфигурированы для работы от имени пользователя и/или группы, отличных от определенных в образе контейнера;
- контейнеры также могут работать в привилегированном режиме, позволяя им получать доступ к устройствам узла, которые в противном случае не будут доступны для модулей;
- контейнеры можно запускать только для чтения, не давая процессам выполнять запись в файловую систему контейнера (и позволяя им выполнять запись только в смонтированные тома);
- ресурсы политики безопасности модуля PodSecurityPolicy на уровне кластера создаются для предотвращения создания пользователями модулей, которые могут поставить под угрозу узел;
- ресурсы политики безопасности модуля могут быть связаны с конкретными пользователями, используя кластерные роли ClusterRole и привязки ClusterRoleBinding на основе RBAC;
- ресурсы политики сети NetworkPolicy используются для лимитирования входящего и/или исходящего трафика модуля.

В следующей главе вы узнаете, как ограничивать доступные модулям вычислительные ресурсы и как конфигурировать качество обслуживания модулей.

Глава 14

Управление вычислительными ресурсами модулей

Эта глава посвящена:

- запросам на ЦП, память и другие вычислительные ресурсы для контейнеров;
- установке жесткого лимита на ЦП и память;
- гарантии качества обслуживания для модулей;
- установке стандартных, минимальных и максимальных ресурсов для модулей в пространстве имен;
- лимитированию общего объема ресурсов, доступных в пространстве имен.

До сих пор вы создавали модули, не заботясь о том, какой объем памяти и ЦП они потребляют. Но, как вы увидите в этой главе, установка ожидаемого объема потребления и максимально разрешенного объема потребления модулем являются важной составной частью любого определения модуля. Установка этих двух наборов параметров гарантирует, что модуль будет получать только свою справедливую долю ресурсов, предоставляемых кластером Kubernetes, а также влияет на то, как модули в кластере назначаются узлам.

14.1 Запрос на ресурсы для контейнеров модуля

При создании модуля вы можете указать объем ЦП и памяти, необходимый контейнеру (они называются *запросами*), а также жесткий лимит на то, что он может потреблять (так называемые *лимиты*). Они задаются для каждого контейнера в отдельности, а не для модуля в целом. Ресурсные запросы и лимиты модуля представляют собой сумму запросов и лимитов всех его контейнеров.

14.1.1 Создание модулей с ресурсными запросами

Рассмотрим пример манифеста модуля, в котором указаны запросы на ЦП и память для одного контейнера, как показано в следующем ниже листинге.

Листинг 14.1. Модуль с ресурсными запросами: requests-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: requests-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:
      requests:
        cpu: 200m
        memory: 10Mi
```

Вы указываете ресурсные запросы для главного контейнера

Контейнер запрашивает 200 миллиядер (то есть 1/5 от времени одного ядра процессора)

Контейнер также запрашивает 10 мебибайт памяти

В манифесте модуля для правильной работы одного контейнера требуется одна пятая часть ядра ЦП (200 миллиядер). Пять таких модулей/контейнеров могут работать достаточно быстро на одном ядре ЦП.

Когда вы не указываете запрос на ЦП, вы говорите, что вам все равно, сколько процессорного времени выделено процессу, работающему в вашем контейнере. В худшем случае он может вообще не получить процессорного времени (это происходит, когда на ЦП существует большой спрос со стороны других процессов). Хотя это может быть хорошо для низкоприоритетных пакетных заданий, которые не являются критичными по времени, очевидно, что это не подходит для контейнеров, обрабатывающих запросы пользователей.

В спецификации модуля также запрашивается 10 мебибайт памяти для контейнера. Поступая так, вы говорите, что ожидаете, что процессы, работающие внутри контейнера, будут использовать не более 10 мебибайт оперативной памяти. Они могут использовать меньше, но вы не ожидаете, что они будут использовать больше, чем в обычных обстоятельствах. Позже в этой главе вы увидите, что произойдет, если они используют больше.

Теперь вы запустите модуль. При запуске модуля можно быстро просмотреть потребление ЦП процессом, выполнив команду `top` внутри контейнера, как показано в следующем ниже листинге.

Листинг 14.2. Исследование потребления ЦП и памяти в контейнере

```
$ kubectl exec -it requests-pod top
Mem: 1288116K used, 760368K free, 9196K shrd, 25748K buff, 814840K cached
```

```
CPU:  9.1% usr 42.1% sys 0.0% nic 48.4% idle 0.0% io 0.0% irq 0.2% sirq
Load average: 0.79 0.52 0.29 2/481 10
  PID PPID USER      STAT   VSZ  %VSZ  CPU  %CPU  COMMAND
    1    0 root      R     1192   0.0    1  50.2  dd if /dev/zero of /dev/null
    7    0 root      R     1200   0.0    0   0.0  top
```

Команда dd, выполняемая в контейнере, потребляет столько ЦП, сколько она может, но она выполняет только один поток, поэтому она может использовать лишь одно ядро. Виртуальная машина Minikube, на которой выполняется данный пример, имеет два выделенных ей ядра ЦП. Вот почему процесс показан как потребляющий 50% всего ЦП.

Пятьдесят процентов от двух ядер – это, разумеется, одно целое ядро, то есть контейнер использует более 200 миллиядер, которые вы запросили в спецификации модуля. Это ожидаемо, поскольку запросы не лимитируют объем ЦП, который может использовать контейнер. Для этого необходимо указать лимит ЦП. Вы попробуете это позже, но сначала давайте посмотрим, как определение ресурсных запросов в модуле влияет на назначение модуля узлу.

14.1.2 Как ресурсные запросы влияют на назначение модуля узлу

Указывая ресурсные запросы, вы указываете минимальное количество ресурсов, необходимых вашему модулю. Эта информация используется планировщиком при назначении модуля узлу. Каждый узел имеет определенный объем ЦП и памяти, который он может выделять для модулей. При назначении модуля узлу планировщик будет учитывать только те узлы, которые имеют достаточный объем нераспределенных ресурсов, необходимый для удовлетворения потребностей модуля в ресурсах. Если объем нераспределенного ЦП или памяти меньше, чем запрашивает модуль, Kubernetes не будет назначать модуль этому узлу, так как узел не может предоставить минимальный объем, требуемый модулем.

Как планировщик определяет, может ли модуль поместиться на узле

Здесь важно и несколько удивительно то, что планировщик смотрит не на объем каждого отдельного ресурса, который используется в конкретное время назначения модуля узлу, а на сумму ресурсов, запрашиваемых существующими модулями, развернутыми на узле. Несмотря на то что существующие модули могут использовать меньше, чем они запросили, назначение еще одного модуля на основе фактического потребления ресурсов нарушит гарантию, предоставленную уже развернутым модулям.

Это показано на рис. 14.1. На узле развернуто три модуля. Вместе они запросили 80% ЦП узла и 60% памяти узла. Модуль D, показанный в правом нижнем углу рисунка, не может быть назначен узлу, потому что он запрашивает 25% ЦП, что больше, чем 20% нераспределенного ЦП. Тот факт, что три модуля в настоящее время используют только 70% процессора, не имеет значения.

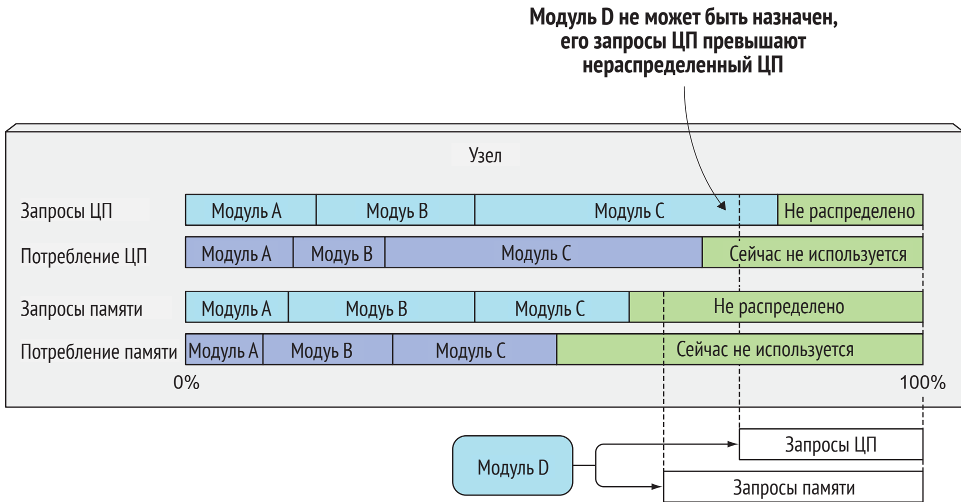


Рис. 14.1. Планировщика интересуют только запросы, а не фактическое использование

Как планировщик использует запросы модулей при выборе наилучшего узла для модуля

Из главы 11 вы, возможно, помните, что планировщик сначала фильтрует список узлов, чтобы исключить те, на которых модуль не может поместиться, а затем приоритизирует оставшиеся узлы в соответствии с настроенными функциями приоритизации. Среди прочего две функции приоритизации ранжируют узлы на основе объема запрашиваемых ресурсов: `LeastRequestedPriority` и `MostRequestedPriority`. Первая предпочитает узлы с меньшим количеством запрашиваемых ресурсов (с большим количеством нераспределенных ресурсов), в то время как вторая – с точностью до наоборот – предпочитает узлы, которые имеют наиболее запрашиваемые ресурсы (наименьший объем нераспределенного ЦП и памяти). Но, как мы уже обсуждали, они обе учитывают объем запрашиваемых ресурсов, а не объем фактически потребляемых ресурсов.

Планировщик настроен на применение только одной из этих функций. Вы можете задаться вопросом, зачем кому-то использовать функцию `MostRequestedPriority`. В конце концов, если у вас есть набор узлов, то вы обычно хотите равномерно распределить по ним нагрузку на процессор. Однако это не относится к облачной инфраструктуре, где при необходимости можно добавлять и удалять узлы. Настроив планировщик на использование функции `MostRequestedPriority`, вы гарантируете, что Kubernetes будет использовать наименьшее возможное количество узлов, предоставляя каждому модулю объем ЦП/памяти, который он запрашивает. Если держать модули плотно упакованными, то некоторые узлы будут оставаться свободными и могут быть удалены. Поскольку вы платите за отдельные узлы, это экономит ваши деньги.

Инспектирование емкости узла

Давайте посмотрим на планировщика в действии. Вы развернете еще один модуль с объемом запрашиваемых ресурсов в четыре раза больше, чем раньше. Но прежде чем вы это сделаете, давайте посмотрим на емкость вашего узла. Поскольку планировщику необходимо знать, сколько ЦП и памяти имеет каждый узел, агент Kubelet передает эти данные на сервер API, делая их доступными через ресурс узла Node. Это можно увидеть с помощью команды `kubectl describe`, как показано в следующем ниже листинге.

Листинг 14.3. Емкость и выделяемые ресурсы узла

```
$ kubectl describe nodes
Name:       minikube
...
Capacity:
  cpu:           2          <- Общая емкость узла
  memory:        2048484Ki
  pods:          110
Allocatable:
  cpu:           2          <- Ресурсы, выделяемые для модулей
  memory:        1946084Ki
  pods:          110
...
```

Результаты показывают два набора объемов, связанных с доступными ресурсами на узле: *емкость* и *выделяемые* ресурсы узла. Емкость представляет общие ресурсы узла, которые не все могут быть доступны для модулей. Некоторые ресурсы могут быть зарезервированы для Kubernetes и/или системных компонентов. Планировщик основывает свои решения только на выделяемых объемах ресурсов.

В предыдущем примере узел с именем `minikube` выполняется в виртуальной машине с двумя ядрами и не имеет зарезервированного ЦП, что приводит к тому, что для модулей выделяется весь ЦП. Поэтому планировщик не должен иметь никаких проблем, назначая еще один модуль, запрашивающий 800 миллиядер.

Теперь запустите этот модуль. Вы можете использовать файл YAML из архива кода либо запустить его с помощью команды `kubectl run`, как показано ниже:

```
$ kubectl run requests-pod-2 --image=busybox --restart Never
➥   --requests='cpu=800m,memory=20Mi' -- dd if=/dev/zero of=/dev/null
pod "requests-pod-2" created
```

Посмотрим, был ли он назначен узлу:

```
$ kubectl get po requests-pod-2
NAME            READY   STATUS    RESTARTS   AGE
requests-pod-2  1/1     Running   0          3m
```

Отлично. Модуль был назначен и работает.

Создание модуля, который не помещается ни на одном узле

Теперь у вас есть два модуля, которые вместе запросили в общей сложности 1000 миллиядер, или ровно 1 ядро. Поэтому вы должны иметь еще 1000 миллиядер для дополнительных модулей, не так ли? Вы можете развернуть еще один модуль с ресурсным запросом на 1000 миллиядер. Используйте команду, аналогичную предыдущей:

```
$ kubectl run requests-pod-3 --image=busybox --restart Never
➥   --requests='cpu=1,memory=20Mi' -- dd if=/dev/zero of=/dev/null
pod "requests-pod-2" created
```

> **ПРИМЕЧАНИЕ.** На этот раз вы указываете запрос на ЦП в целых ядрах (`cpu=1`) вместо миллиядер (`cpu=1000m`).

Пока все хорошо. Модуль был принят сервером API (из предыдущей главы вы помните, что сервер API может отклонить модули, если они недействительны по той или иной причине). Теперь проверьте, работает ли модуль:

```
$ kubectl get po requests-pod-3
NAME            READY  STATUS   RESTARTS  AGE
requests-pod-3  0/1    Pending  0         4m
```

Даже если вы подождете некоторое время, модуль по-прежнему будет висеть в состоянии ожидания. Дополнительную информацию о том, почему так происходит, можно получить с помощью команды `kubectl describe`, как показано в следующем ниже листинге.

Листинг 14.4. Исследование, почему модуль застрял в ожидании, с помощью команды kubectl describe pod

```
$ kubectl describe po requests-pod-3
Name:        requests-pod-3
Namespace:   default
Node:        /                          <- Ни один узел не связан
...                                        с модулем
Conditions:
Type             Status
PodScheduled     False                  <- Модуль не был
...                                        назначен
Events:
... Warning FailedScheduling    No nodes are available
                                that match all of the
                                following predicates::     <- Назначение модуля
                                Insufficient cpu (1).         узлу не удалось из-за
                                                              недостаточного объема ЦП
```

Результаты показывают, что модуль не был назначен, потому что он не может поместиться ни на одном узле из-за недостаточного объема ЦП на вашем одиночном узле. Но почему? Сумма запросов ЦП всех трех модулей равна 2000 миллиядрам, или ровно двум ядрам, то есть именно тому, что может предоставить ваш узел. Что же случилось?

Почему модуль не назначается узлу

Вы можете выяснить, почему модуль не был назначен узлу, проинспектировав ресурс узла. Примените команду `kubectl describe node` еще раз и внимательно изучите результаты в следующем ниже листинге.

Листинг 14.5. Инспектирование выделенных ресурсов на узле с помощью команды kubectl describe node

```
$ kubectl describe node
Name:                   minikube
...
Non-terminated Pods:    (7 in total)
  Namespace      Name            CPU Requ.   CPU Lim.  Mem Req.    Mem Lim.
  ---------      ----            ----------  --------  ---------   --------
  default        requests-pod    200m (10%)  0 (0%)    10Mi (0%)   0 (0%)
  default        requests-pod-2  800m (40%)  0 (0%)    20Mi (1%)   0 (0%)
  kube-system    dflt-http-b...  10m (0%)    10m (0%)  20Mi (1%)   20Mi (1%)
  kube-system    kube-addon-...  5m (0%)     0 (0%)    50Mi (2%)   0 (0%)
  kube-system    kube-dns-26...  260m (13%)  0 (0%)    110Mi (5%)  170Mi (8%)
  kube-system    kubernetes-...  0 (0%)      0 (0%)    0 (0%)      0  (0%)
  kube-system    nginx-ingre...  0 (0%)      0 (0%)    0 (0%)      0  (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits      Memory Requests Memory Limits
  ------------  ----------      --------------- -------------
  1275m (63%)   10m (0%)        210Mi (11%)     190Mi (9%)
```

Если вы посмотрите в левый нижний угол списка, то увидите, что в общей сложности работающими модулями было запрошено 1275 миллиядер, что на 275 миллиядер больше, чем вы запросили для первых двух модулей, которые вы развернули. Что-то съедает дополнительные ресурсы процессора.

Вы можете найти виновника в списке модулей в предыдущем листинге. Три модуля в пространстве имен `kube-system` явно запрашивали ресурсы ЦП. Эти модули плюс два ваших модуля оставляют для дополнительных модулей только 725 миллиядер. Поскольку ваш третий модуль запросил 1000 миллиядер, планировщик не будет назначать его этому узлу, так как это сделает узел перегруженным.

Освобождение ресурсов, чтобы модуль был назначен узлу

Модуль будет назначен только при освобождении достаточного объема ЦП (например, при удалении одного из первых двух модулей). Если вы удалите

второй модуль, то планировщик будет уведомлен об удалении (через механизм наблюдения, описанный в главе 11) и назначит узлу третий модуль, как только второй модуль завершит работу. Это показано в следующем ниже листинге.

Листинг 14.6. Модуль назначен после удаления другого модуля

```
$ kubectl delete po requests-pod-2
pod «requests-pod-2» deleted

$ kubectl get po
NAME            READY   STATUS        RESTARTS   AGE
requests-pod    1/1     Running       0          2h
requests-pod-2  1/1     Terminating   0          1h
requests-pod-3  0/1     Pending       0          1h

$ kubectl get po
NAME            READY   STATUS    RESTARTS   AGE
requests-pod    1/1     Running   0          2h
requests-pod-3  1/1     Running   0          1h
```

Во всех этих примерах вы указывали запрос на память, но он не играл никакой роли в назначении, потому что ваш узел имеет более чем достаточно выделяемой памяти для размещения всех запросов в ваших модулях. Запросы на ЦП и память обрабатываются планировщиком одинаково, но, в отличие от запросов на память, запросы модуля на ЦП, кроме того, играют свою роль еще в одном месте – во время работы модуля. Вы узнаете об этом далее.

14.1.3 Как запросы на ЦП влияют на совместное использование процессорного времени

Теперь у вас в вашем кластере работает два модуля (прямо сейчас вы можете проигнорировать системные модули, потому что они в основном простаивают). Один запросил 200 миллиядер, другой – в пять раз больше. В начале главы мы отметили, что в Kubernetes есть различие между ресурсными запросами и лимитами. Вы еще не определили никаких лимитов, поэтому эти два модуля никоим образом не лимитированы, когда дело касается того, какой объем ЦП они могут потреблять. Если процесс внутри каждого модуля потребляет столько процессорного времени, сколько может, тогда вопрос состоит в том, сколько процессорного времени получает каждый модуль.

Запросы на ЦП не только влияют на назначение модулей узлам – они также определяют, как оставшееся (неиспользуемое) процессорное время распределяется между модулями. Поскольку ваш первый модуль запросил 200 миллиядер ЦП, а другой – 1000 миллиядер, то любой неиспользуемый ЦП будет разделен между двумя модулями в соотношении 1 к 5, как показано на рис. 14.2. Если оба модуля потребляют столько процессора, сколько они могут, то пер-

вый модуль получит одну шестую, или 16,7%, процессорного времени, а другой – оставшиеся пять шестых, или 83,3%.

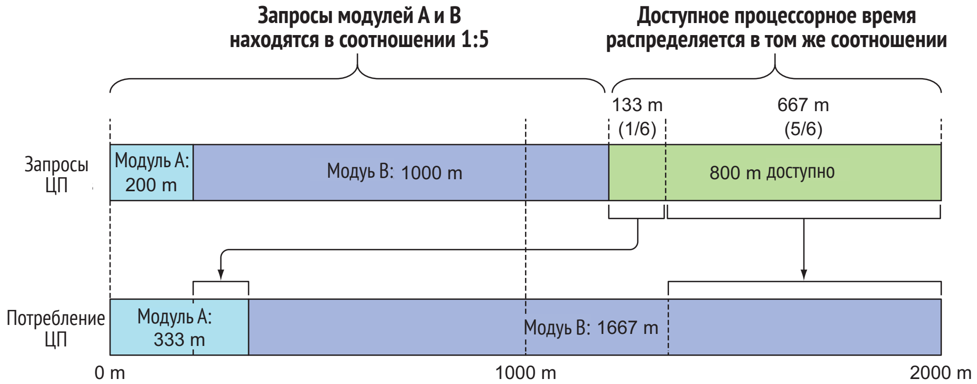


Рис. 14.2. Процессорное время распределяется среди контейнеров с учетом их запросов на ЦП

Но если один контейнер хочет использовать как можно больше ЦП, в то время как другой в данный момент находится в режиме ожидания, первому контейнеру будет разрешено использовать все время ЦП (за вычетом небольшого количества времени, используемого вторым контейнером, если таковое имеется). В конце концов, если никто не использует ЦП, имеет смысл использовать весь доступный ЦП, не так ли? Как только второму контейнеру потребуется процессорное время, он его получит, и первый контейнер будет снова отрегулирован.

14.1.4 Определение и запрос настраиваемых ресурсов

Kubernetes также позволяет добавлять в узел собственные настраиваемые ресурсы и запрашивать их в ресурсных запросах модуля. Первоначально они назывались непрозрачными целочисленными ресурсами (Opaque Integer Resources), но в версии 1.8 Kubernetes были заменены расширенными ресурсами (Extended Resources).

Прежде всего вам, очевидно, нужно поставить Kubernetes в известность о вашем, созданном пользователем ресурсе, добавив его в поле `capacity` объекта Node. Это можно сделать, выполнив HTTP-запрос `PATCH`. Имя ресурса может быть любым, например `example.org/my-resource`, – любым до той поры, пока он не начинается с домена `kubernetes.io`. Заданное количество должно быть целым числом (например, вы не можете установить его в 100 миллиединицах, потому что 0.1 не является целым числом; но вы можете установить его в 1000 m или 2000 m, или, просто, 1 или 2). Это значение будет автоматически скопировано из поля `capacity` в поле `allocatable`.

Затем при создании модулей вам нужно указать в поле `resources.requests` в секции `spec` контейнера то же самое имя ресурса и запрошенное количество или же сделать это с помощью параметра `--requests` при использовании ко-

манды `kubectl run` так же, как и в предыдущих примерах. Планировщик прогарантирует, что модуль будет развернут только на том узле, который имеет запрошенный объем доступного, созданного пользователем ресурса. Каждый развернутый модуль, безусловно, уменьшает количество выделяемых единиц ресурса.

Примером настраиваемого ресурса может быть число доступных на узле единиц GPU. Модули, требующие использования GPU, указывают это в своих запросах. Планировщик затем удостоверяется, что модуль назначается узлам, по крайней мере, только с одним еще не распределенным GPU.

14.2 Лимитирование ресурсов, доступных контейнеру

Установка ресурсных запросов для контейнеров в модуле гарантирует, что каждый контейнер получит минимальный объем необходимых ему ресурсов. Теперь давайте посмотрим на другую сторону монеты – на максимальный объем, который контейнеру будет разрешено потреблять.

14.2.1 Установка жесткого лимита на объем ресурсов, которые может использовать контейнер

Мы видели, как контейнерам разрешено использовать весь ЦП, если все другие процессы находятся в режиме ожидания. Но может возникнуть ситуация, когда некоторым контейнерам потребуется запретить использовать больше, чем конкретный объем ЦП. И вам всегда будет требоваться лимитировать объем памяти, который контейнер может потреблять.

ЦП – это сжимаемый ресурс, имейте в виду, что используемый контейнером объем может регулироваться без неблагоприятного влияния на работающий в контейнере процесс. Память, очевидным образом, отличается: она несжимаема. После того как процесс получает блок памяти, эта память не может быть из него удалена, пока она не будет освобождена самим процессом. Вот почему вам нужно лимитировать максимальный объем памяти, предоставляемый контейнеру.

Не лимитируя память, контейнер (или модуль), работающий на рабочем узле, может съесть всю доступную память и повлиять на все другие модули на узле и любые новые модули, назначаемые узлу (напомним, что новые модули назначаются узлу на основе запросов на память, а не на основе фактического потребления памяти). Один неисправный или вредоносный модуль может практически привести весь узел в непригодность.

Создание модуля с лимитами на ресурсы

Для того чтобы этого не произошло, Kubernetes позволяет для каждого контейнера задавать лимиты на ресурсы (наряду и практически так же, как и с

ресурсными запросами). В следующем ниже листинге показан пример манифеста модуля с лимитами на ресурсы.

Листинг 14.7. Модуль с жестким ограничением на ЦП и память: limited-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: limited-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:              <- Указание лимитов на ресурсы для контейнера
      limits:
        cpu: 1              <- Этому контейнеру будет разрешено использовать не более 1 ядра ЦП
        memory: 20Mi        <- Контейнеру будет разрешено использовать до 20 мебибайт памяти
```

В контейнере этого модуля настроены лимиты на ресурсы как для ЦП, так и для памяти. Процесс или процессы, работающие внутри контейнера, не смогут потреблять более 1 ядра ЦП и 20 мебибайт памяти.

ПРИМЕЧАНИЕ. Поскольку вы не указали ресурсные запросы, они будут иметь те же значения, что и ресурсные лимиты.

Перегрузка лимитов

В отличие от ресурсных запросов, ресурсные лимиты не ограничены выделяемыми объемами ресурсов узла. Сумме всех лимитов всех модулей на узле разрешено превышать 100% емкости узла (рис. 14.3). При пересчете лимиты на ресурсы могут быть перегружены. Это имеет важное последствие – когда 100% ресурсов узла израсходованы, определенные контейнеры должны быть уничтожены.

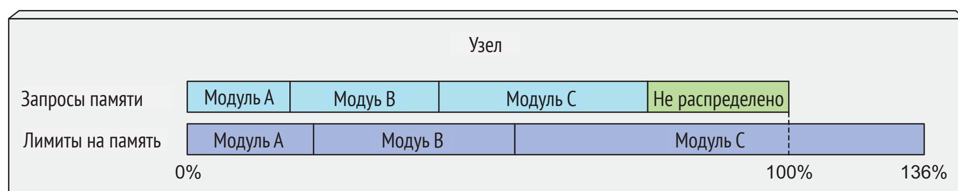


Рис. 14.3. Сумма лимитов ресурсов всех модулей на узле может превышать 100% мощности узла

В разделе 14.3 вы увидите, каким образом Kubernetes решает, какие контейнеры уничтожать, но отдельные контейнеры могут быть уничтожены, даже

если они пытаются использовать больше, чем указано в их лимитах на ресурсы. Далее вы узнаете об этом подробнее.

14.2.2 Превышение лимитов

Что происходит, когда процесс, работающий в контейнере, пытается использовать больший объем ресурсов, чем разрешено?

Вы уже узнали, что ЦП является сжимаемым ресурсом, и вполне естественно, что процесс хочет потреблять все процессорное время, не дожидаясь операции ввода-вывода. Как вы узнали, использование ЦП процессом регулируется, поэтому, когда для контейнера задан лимит на ЦП, процесс не получает большее процессорное время, чем сконфигурированный лимит.

С памятью все по-другому. Когда процесс пытается выделить память сверх своего лимита, процесс уничтожается (говорят, что контейнер убивается из-за OOM, где OOM обозначает нехватку памяти, Out Of Memory). Если политика перезапуска модуля задана как `Always` или `OnFailure`, то процесс перезапускается немедленно, поэтому вы можете даже не заметить, что он был уничтожен. Но если он продолжает выходить за пределы памяти и уничтожаться, то Kubernetes начнет его перезапуск с увеличивающимися задержками между перезапусками. В этом случае вы увидите статус цикличного отката `CrashLoopBackOff`:

```
$ kubectl get po
NAME       READY  STATUS            RESTARTS  AGE
memoryhog  0/1    CrashLoopBackOff  3         1m
```

Статус `CrashLoopBackOff` не означает, что агент Kubelet сдался. Он означает, что после каждого фатального сбоя Kubelet увеличивает период времени до перезапуска контейнера. После первого фатального сбоя он перезапускает контейнер немедленно, и затем, если контейнер снова сбоит, перед повторным перезапуском он ждет 10 секунд. На последующих фатальных ситуациях эта задержка потом увеличивается до 20, 40, 80 и 160 секунд и, наконец, ограничивается 300 секундами. Как только интервал достигает 300-секундного предела, Kubelet продолжает бесконечно перезапускать контейнер каждые пять минут до тех пор, пока модуль не остановит фатальный сбой либо не будет удален.

Для того чтобы исследовать причину фатального сбоя контейнера, можно проверить журнал модуля и/или применить команду `kubectl describe pod`, как показано в следующем ниже листинге.

Листинг 14.8. Инспектирование причины, почему контейнер прекращает работу, с помощью команды kubectl describe pod

```
$ kubectl describe pod
Name:       memoryhog
...
```

```
Containers:
  main:
    ...
    State:          Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Tue, 27 Dec 2016 14:55:53 +0100
      Finished:     Tue, 27 Dec 2016 14:55:58 +0100
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Tue, 27 Dec 2016 14:55:37 +0100
      Finished:     Tue, 27 Dec 2016 14:55:50 +0100
    Ready:          False
...
```

Текущий контейнер был уничтожен из-за нехватки памяти (OOM)

Предыдущий контейнер также был уничтожен из-за OOM

Статус `OOMKilled` говорит о том, что контейнер был уничтожен, потому что он испытывал нехватку памяти. В приведенном выше листинге контейнер превысил лимит памяти и был немедленно уничтожен.

Важно не устанавливать слишком низкие лимиты памяти, если вы не хотите, чтобы ваш контейнер был уничтожен. Но контейнеры могут быть уничтожены, даже если они не превышают своего лимита. Вы увидите причину этому в разделе 14.3.2, но сначала давайте обсудим нечто, что заставляет врасплох большинство пользователей, когда они в первый раз начинают устанавливать лимиты для своих контейнеров.

14.2.3 Как приложения в контейнерах видят лимиты

Если вы не развернули модуль из листинга 14.7, то разверните его сейчас:

```
$ kubectl create -f limited-pod.yaml
pod "limited-pod" created
```

Теперь выполните команду `top` в контейнере, как вы это сделали в начале главы. Результаты команды показаны в следующем ниже листинге.

Листинг 14.9. Выполнение команды top в контейнере, лимитированном по ЦП и памяти

```
$ kubectl exec -it limited-pod top
Mem: 1450980K used, 597504K free, 22012K shrd, 65876K buff, 857552K cached
CPU: 10.0% usr 40.0% sys 0.0% nic 50.0% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 0.17 1.19 2.47 4/503 10
  PID  PPID  USER     STAT   VSZ  %VSZ CPU %CPU COMMAND
    1     0  root     R      1192  0.0   1 49.9 dd if /dev/zero of /dev/null
    5     0  root     R      1196  0.0   0  0.0 top
```

Прежде всего следует напомнить, что лимит на ЦП модуля установлен в 1 ядро, и его лимит на память установлен в 20 MiB. Теперь внимательно изучите вывод команды `top`. Есть ли что-то, что кажется вам странным?

Посмотрите на объем используемой и свободной памяти. Эти цифры далеко не рядом с 20 MiB, которые вы установили в качестве лимита для контейнера. Точно так же вы установили лимит на ЦП в одно ядро, и, похоже, главный процесс использует только 50% доступного процессорного времени, даже если команда dd, при использовании, как вы используете ее, обычно использует весь ЦП, который она имеет в наличии. Что же происходит?

Контейнеры всегда видят память узла, а не контейнера

Команда `top` показывает объем памяти всего узла, на котором работает контейнер. Несмотря на то что задан лимит на объем памяти, доступной для контейнера, контейнер не будет осведомлен об этом лимите.

Это оказывает неблагоприятное влияние на любое приложение, которое ищет объем доступной памяти в системе и использует эту информацию, чтобы решить, сколько памяти ей зарезервировать.

Данная проблема видна при выполнении приложений Java, в особенности если вы не указываете максимальный размер кучи для виртуальной машины Java с помощью параметра `-Xmx`. В этом случае JVM установит максимальный размер кучи не на основе доступной контейнеру памяти, а на основе общего объема памяти узла. При запуске контейнерных приложений Java в кластере Kubernetes на ноутбуке эта проблема не проявляется, поскольку разница между установленными для модуля лимитами на память и общим объемом памяти, доступным на ноутбуке, не так велика.

Но при развертывании модуля в рабочем окружении, где узлы имеют гораздо больше физической памяти, JVM может превысить сконфигурированный вами лимит на память контейнера и будет уничтожен из-за нехватки памяти (`OOMKilled`).

И если вы думаете, что правильная установка параметра `-Xmx` решает данную проблему, то вы, к сожалению, ошибаетесь. Параметр `-Xmx` ограничивает только размер кучи, но ничего не делает с памятью вне кучи JVM. К счастью, новые версии Java эту проблему облегчают, учитывая сконфигурированные лимиты контейнера.

Контейнеры также видят все ядра ЦП узла

Точно так же, как с памятью, контейнеры также будут видеть все ЦП узла, независимо от лимитов на ЦП, сконфигурированных для контейнера. Установка лимита на ЦП на одно ядро не предоставляет контейнеру волшебным образом доступа только к одному ядру ЦП. Лимит на ЦП делает лишь то, что ограничивает количество процессорного времени, которое контейнер может использовать.

Контейнер с одноядерным лимитом на ЦП, работающий на 64-ядерном ЦП, получит 1/64 часть общего процессорного времени. И хотя его лимит

установлен в одно ядро, процессы контейнера не будут выполняться только на одном ядре. В разные моменты времени его код может выполняться на разных ядрах.

В этом нет ничего плохого, верно? Хотя это в целом так, существует, по крайней мере, один сценарий, когда эта ситуация катастрофична.

Некоторые приложения ищут объем ЦП в системе, чтобы определить, сколько рабочих потоков они должны выполнять. Опять же, такое приложение будет отлично работать на ноутбуке для разработки, но при развертывании на узле с гораздо большим количеством ядер оно будет запускать слишком много потоков, и все будут конкурировать за (возможно) лимитированное процессорное время. Кроме того, каждый поток требует дополнительной памяти, в результате чего использование памяти приложений стремительно растет.

Вы можете использовать downward API, чтобы передать лимит на ЦП контейнеру и использовать его, не опираясь на объем ЦП, который ваше приложение может видеть в системе. Вы также можете напрямую посмотреть, что происходит в системе cgroups, чтобы получать настроенный лимит на ЦП, прочитав следующие ниже файлы:

- /sys/fs/cgroup/cpu/cpu.cfs_quota_us;
- /sys/fs/cgroup/cpu/cpu.cfs_period_us.

14.3 Классы QoS модулей

Мы уже отмечали, что лимиты на ресурсы могут быть перегружены и что узел не может предоставлять всем своим модулям объем ресурсов, указанный в их лимитах на ресурсы.

Представьте, что у вас есть два модуля, где модуль A использует, скажем, 90% памяти узла, и потом модуль B внезапно требует памяти больше, чем тот объем, который он использовал до этого момента, и узел не может обеспечить необходимый объем памяти. Какой контейнер должен быть уничтожен? Должен ли это быть модуль B, потому что его запрос на память не может быть удовлетворен, либо должен быть уничтожен модуль A, чтобы освободить память, которую можно было бы предоставить модулю B?

Совершенно очевидно, что все зависит от конкретной ситуации. Kubernetes не может принимать правильное решение самостоятельно. Вам нужен способ указывать, какие модули в таких случаях имеют приоритет. Kubernetes делает это, классифицируя модули на три класса качества обслуживания (QoS):

- `BestEffort` (самый низкий приоритет);
- `Burstable`;
- `Guaranteed` (самый высокий).

14.3.1 Определение класса QoS для модуля

Вы, возможно, ожидаете, что эти классы будут назначаться модулям через отдельное поле в манифесте, но это не так. Класс QoS является производным

от комбинации ресурсных запросов и лимитов для контейнеров модуля. И вот как это делается.

Назначение модуля классу BestEffort

Класс QoS с самым низким приоритетом – это класс `BestEffort`. Он назначается модулям, у которых вообще нет запросов или лимитов (ни в одном из их контейнеров). Этот класс QoS назначается всем модулям, созданным в предыдущих главах. Контейнеры, работающие в этих модулях, не имели никаких гарантий ресурсов. В худшем случае они могут совсем не получить почти никакого процессорного времени и будут уничтожены первыми, когда требуется освободить память для других модулей. Но поскольку модуль `BestEffort` не имеет установленных лимитов на память, его контейнеры могут использовать столько памяти, сколько они хотят, если в наличии имеется достаточно памяти.

Назначение модуля классу Guaranteed

На другом конце спектра находится класс QoS `Guaranteed`. Этот класс дается модулям, в которых запросы контейнеров равны лимитам на все ресурсы. Для того чтобы класс модуля был `Guaranteed`, должны быть истинными три вещи:

- запросы и лимиты должны быть установлены как для процессора, так и для памяти;
- их нужно установить для каждого контейнера;
- они должны быть равными (лимит должен соответствовать запросу для каждого ресурса в каждом контейнере).

Поскольку ресурсные запросы контейнера, в случае если они не заданы явно, по умолчанию равны лимитам, установление лимитов на все ресурсы (для каждого контейнера в модуле) будет достаточно для того, чтобы модуль был прогарантированным, то есть принадлежал классу `Guaranteed`. Контейнеры в этих модулях получают запрошенный объем ресурсов, но не могут потреблять дополнительные (поскольку их лимиты не превышают их запросы).

Назначение модуля классу Burstable

Между классами `BestEffort` и `Guaranteed` расположен класс QoS `Burstable`. Все остальные модули попадают в этот класс. Сюда включаются одноконтейнерные модули, где лимиты контейнера не соответствуют его запросам, и все модули, где хотя бы один контейнер имеет указанный ресурсный запрос, но не лимит. Сюда также включаются модули, в которых запросы одного контейнера соответствуют их лимитам, но в другом контейнере не указаны запросы или лимиты. Модули `Burstable` получают объем ресурсов, который они запрашивают, но им разрешено, если это требуется, использовать дополнительные ресурсы (вплоть до лимита).

Как связь между запросами и лимитами определяет класс QoS

Все три класса QoS и их связи с запросами и лимитами показаны на рис. 14.4.

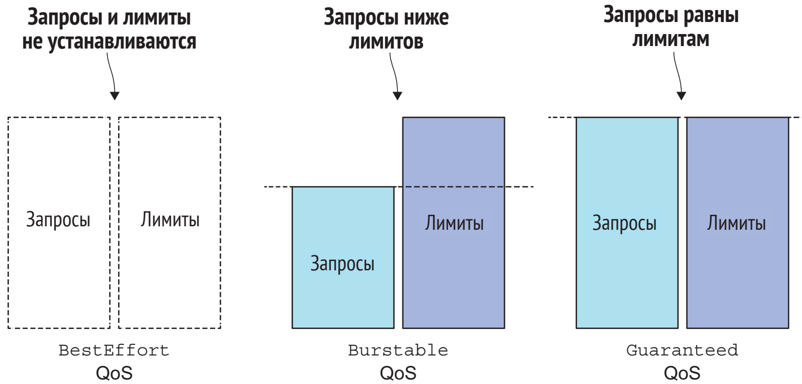


Рис. 14.4. Ресурсные запросы, лимиты и классы QoS

От размышлений о том, какой класс QoS имеет модуль, ваша голова может пойти кругом, потому что этот вопрос связан с несколькими контейнерами, несколькими ресурсами и всеми возможными связями между запросами и лимитами. Будет легче, если вы начнете думать о QoS на уровне контейнера (хотя классы QoS являются свойством модулей, а не контейнеров), а затем спроецируете свое понимание классов QoS контейнеров на класс QoS модуля.

Выяснение класса QoS контейнера

Таблица 14.1 показывает класс QoS на основе того, как ресурсные запросы и лимиты определены на одиночном контейнере. В случае одноконтейнерных модулей класс QoS также применим и к модулю.

Таблица 14.1. Класс QoS одноконтейнерного модуля на основе ресурсных запросов и лимитов

Запросы и лимиты на ЦП	Запросы и лимиты на память	Класс QoS контейнера
Оба не установлены	Оба не установлены	`BestEffort`
Оба не установлены	Запросы < Лимиты	`Burstable`
Оба не установлены	Запросы = Лимиты	`Burstable`
Запросы < Лимиты	Оба не установлены	`Burstable`
Запросы < Лимиты	Запросы < Лимиты	`Burstable`
Запросы < Лимиты	Запросы = Лимиты	`Burstable`
Запросы = Лимиты	Запросы = Лимиты	`Guaranteed`

ПРИМЕЧАНИЕ. Если установлены только запросы и нет лимитов, то обратитесь к тем строкам таблицы, в которых запросы меньше лимитов. Если же установлены только лимиты, а запросы по умолчанию равны лимитам, тогда обратитесь к строкам, где запросы равны лимитам.

Выяснение класса QoS многоконтейнерного модуля

Для мультиконтейнерных модулей, если все контейнеры имеют один и тот же класс QoS, этот класс также является классом QoS модуля. Если, по крайней мере, один контейнер имеет другой класс, то классом QoS модуля является `Burstable`, независимо от того, каковы классы контейнеров. В табл. 14.2 показано, как класс QoS двухконтейнерного модуля связан с классами его двух контейнеров. Вы можете легко расширить это до модулей с более чем двумя контейнерами.

Таблица 14.2. Класс QoS модуля является производным от классов его контейнеров

Класс QoS контейнера 1	Класс QoS контейнера 2	Класс QoS модуля
`BestEffort`	`BestEffort`	`BestEffort`
`BestEffort`	`Burstable`	`Burstable`
`BestEffort`	`Guaranteed`	`Burstable`
`Burstable`	`Burstable`	`Burstable`
`Burstable`	`Guaranteed`	`Burstable`
`Guaranteed`	`Guaranteed`	`Guaranteed`

> **ПРИМЕЧАНИЕ.** Класс QoS модуля показывается при выполнении команды `kubectl describe pod` и в манифесте YAML/JSON модуля в поле `status.qosClass`.

Мы объяснили, как определяются классы QoS, но нам все еще нужно посмотреть, каким образом они определяют, какой контейнер должен быть уничтожен в перегруженной системе.

14.3.2 Какой процесс уничтожается при нехватке памяти

Когда система перегружена, классы QoS определяют, какой контейнер уничтожается первым, чтобы освобожденные ресурсы могли быть переданы более высокоприоритетным модулям. Первыми в очереди на уничтожение являются модули в классе `BestEffort`, затем модули `Burstable` и, наконец, модули `Guaranteed`, которые уничтожаются, только если в памяти нуждаются системные процессы.

Как выстраиваются классы QoS

Давайте посмотрим на примере, показанном на рис. 14.5. Представьте, что у вас два одиночных контейнера, где первый имеет класс QoS `BestEffort`, а второй – класс `Burstable`. Когда вся память узла уже исчерпана и один из процессов на узле пытается выделить дополнительную память, для того чтобы выполнить запрос на выделение, системе нужно будет уничтожить один из процессов (возможно, даже процесс, пытающийся выделить дополнительную

память). В этом случае процесс, работающий в модуле `BestEffort`, всегда будет уничтожен до процесса в модуле `Burstable`.

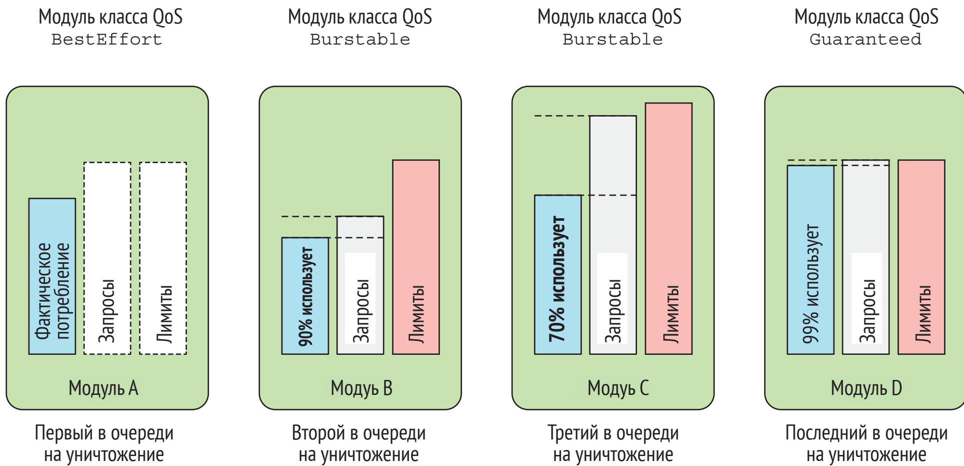


Рис. 14.5. Какие модули уничтожаются первыми

Совершенно очевидно, что процесс модуля `BestEffort` будет также уничтожен до того, как будут уничтожены любые процессы модулей `Guaranteed`. Кроме того, процесс модуля `Burstable` также будет уничтожен до процесса модуля `Guaranteed`. Но что произойдет, если в наличии только два модуля `Burstable`? Очевидно, что процесс отбора должен отдавать предпочтение одному из них.

Как обрабатываются контейнеры с одинаковым классом QoS

Каждый работающий процесс имеет оценку OutOfMemory (OOM). Система выбирает процесс для уничтожения путем сравнения оценок всех работающих процессов. Когда память должна быть освобождена, уничтожается процесс с наибольшей оценкой.

Оценки OOM вычисляются из двух факторов: процента доступной памяти, которую потребляет процесс, и фиксированной корректировки оценки OOM, которая основана на классе QoS модуля и запрошенной памяти контейнера. Когда существуют два одноконтейнерных модуля, оба в классе `Burstable`, система уничтожит тот, который в процентном соотношении использует больше своей запрошенной памяти, чем другой. Вот почему на рис. 14.5 модуль B, использующий 90% запрашиваемой памяти, уничтожается до модуля C, который использует всего 70%, хотя он применяет больше мегабайт памяти, чем модуль B.

Это показывает, что необходимо помнить не только о связи между запросами и лимитами, но и о запросах и ожидаемом фактическом потреблении памяти.

14.4 Установка стандартных запросов и лимитов для модулей в расчете на пространство имен

Мы рассмотрели, как можно устанавливать ресурсные запросы и лимиты для каждого отдельного контейнера. Если вы их не зададите, то контейнер будет находиться во власти всех других контейнеров, которые устанавливают ресурсные запросы и лимиты. Рекомендуется устанавливать запросы и лимиты для каждого контейнера.

14.4.1 Знакомство с ресурсом LimitRange

Вместо того чтобы делать это для каждого контейнера, вы также можете сделать это, создав ресурс диапазона лимитов LimitRange. Он позволяет указывать (для каждого пространства имен) не только минимальный и максимальный лимит, который можно установить для контейнера по каждому ресурсу, но и стандартные ресурсные запросы для контейнеров, которые не устанавливают запросы явным образом, как показано на рис. 14.6.

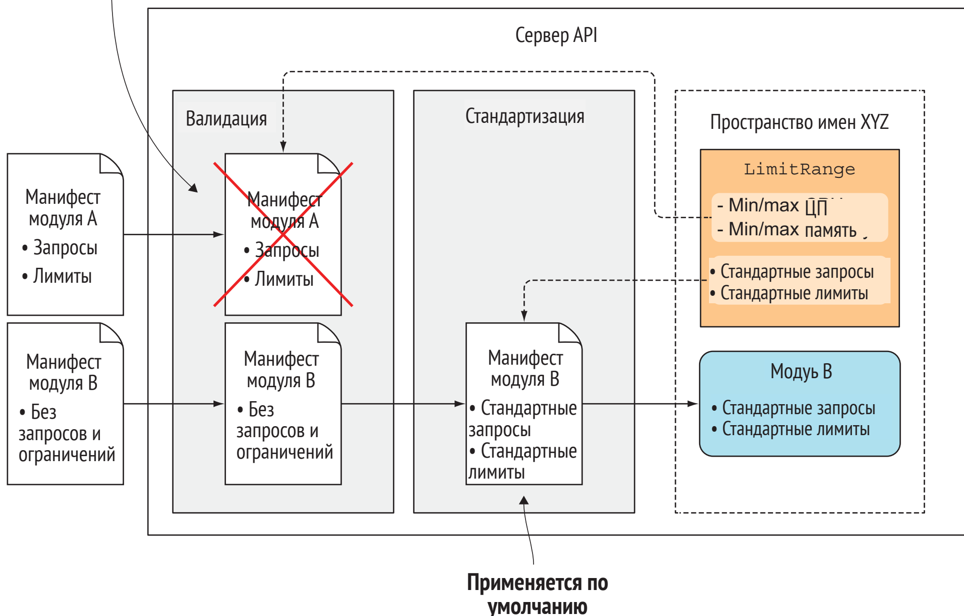


Рис. 14.6. Ресурс LimitRange используется для валидации и стандартизации модулей

Ресурсы LimitRange используются плагином управления допуском (Admission Control plugin) LimitRanger (мы объяснили, что из себя представляют эти плагины, в главе 11). Когда манифест модуля отправляется на сервер API, плагин LimitRanger выполняет проверку допустимости спецификации

модуля. Если валидация не проходит, то манифест немедленно отклоняется. По этой причине отличным вариантом использования объектов LimitRange является недопущение создания пользователями модулей, которые больше любого узла в кластере. Без объекта LimitRange сервер API с радостью примет модуль, но затем никогда не сможет назначить его узлу.

Лимиты, указанные в ресурсе LimitRange, применяются к каждому отдельному модулю/контейнеру или другому объекту, созданному в том же пространстве имен, что и объект LimitRange. Они не ограничивают общий объем ресурсов, доступных во всех модулях пространства имен. Это определяется посредством объектов ResourceQuota, которые объяснены в разделе 14.5.

14.4.2 Создание объекта LimitRange

Давайте рассмотрим полный пример ресурса LimitRange и посмотрим, что делают его отдельные свойства. В следующем ниже листинге показано полное определение ресурса LimitRange.

Листинг 14.10. Ресурс LimitRange: limits.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: example
spec:
  limits:
  - type: Pod
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
  - type: Container
    defaultRequest:
      cpu: 100m
      memory: 10Mi
    default:
      cpu: 200m
      memory: 100Mi
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
```

Задает лимиты для модуля в целом

Минимальные ЦП и память, которые могут запросить контейнеры всех модулей в общей сложности

Максимальные ЦП и память, которые могут запросить (и лимитировать) контейнеры всех модулей

Лимиты контейнера указаны ниже этой строки

Стандартные запросы ЦП и памяти, которые будут применяться к контейнерам, в которых это не указано явно

Стандартные лимиты для контейнеров, в которых это не указано

Минимальные и максимальные запросы/лимиты, которые может иметь контейнер

```
    maxLimitRequestRatio:
      cpu: 4
      memory: 10
  - type: PersistentVolumeClaim
    min:
      storage: 1Gi
    max:
      storage: 10Gi
```

Максимальное соотношение между лимитом и запросом на каждый ресурс

LimitRange может также установить минимальный и максимальный объем хранилища, которое может быть запрошено в заявке PVC

Как видно из предыдущего примера, минимальные и максимальные лимиты для всего модуля могут конфигурироваться. Они применяются к сумме всех запросов и лимитов контейнеров модуля.

Ниже, на уровне контейнера, вы можете установить не только минимум и максимум, но и стандартные ресурсные запросы (`defaultRequest`) и стандартные лимиты (`default`), которые будут применяться к каждому контейнеру, который не указывает их явно.

Помимо значений `min`, `max` и `default`, вы можете даже установить максимальное соотношение лимитов и запросов. В приведенном выше листинге для поля `cpu` под свойством `maxLimitRequestRatio` задано значение 4, имея в виду, что лимиты на ЦП в контейнере не могут быть более чем в четыре раза выше запросов на ЦП. Контейнер, запрашивающий 200 миллиядер, не будет принят, если его лимит ЦП установлен на 801 миллиядро или выше. В случае памяти максимальный коэффициент равен 10.

В главе 6 мы рассмотрели заявки на получение постоянных томов PersistentVolumeClaim (PVC), которые позволяют претендовать на определенный объем постоянного хранилища аналогично тому, как контейнеры модуля требуют ЦП и память. Точно так же, как вы ограничиваете минимальный и максимальный объем ЦП, который может запросить контейнер, вы также должны ограничивать объем хранилища, который может запросить одна заявка PVC. Объект LimitRange это тоже позволяет делать, как показано в нижней части данного примера.

Этот пример показывает один-единственный объект LimitRange, содержащий лимиты для всех объектов, но вы также можете разделить их на несколько объектов, если вы предпочитаете, чтобы они были организованы по типу (к примеру, один для лимитов модуля, другой для лимитов контейнера и еще один для заявок PVC). Лимиты из нескольких объектов LimitRange консолидируются во время валидации модуля или заявки PVC.

Валидация (и стандартные значения), сконфигурированная в объекте LimitRange, выполняется посредством сервера API, когда он получает новый манифест модуля или заявки PVC, поэтому если вы впоследствии измените лимиты, то повторная валидация существующих модулей и заявок PVC выполняться не будет – новые лимиты будут действовать только в модулях и заявках PVC, созданных позже.

14.4.3 Обеспечение лимитов

Теперь, когда лимиты находятся на своем месте, вы можете попробовать создать модуль, который запрашивает ЦП больше, чем разрешено объектом LimitRange. Вы найдете YAML для этого модуля в архиве кода. В следующем ниже листинге показана лишь его часть, относящаяся к данному обсуждению.

Листинг 14.11. Модуль с запросами на ЦП выше лимита: limits-pod-too-big.yaml

```
resources:
  requests:
    cpu: 2
```

Единственный контейнер модуля запрашивает два ЦП, что больше максимального значения, заданного ранее в ресурсе LimitRange. Создание модуля дает следующий результат:

```
$ kubectl create -f limits-pod-too-big.yaml
Error from server (Forbidden): error when creating "limits-pod-too-big.yaml":
pods "too-big" is forbidden: [
  maximum cpu usage per Pod is 1, but request is 2.,
  maximum cpu usage per Container is 1, but request is 2.]
```

Я немного видоизменил результат, чтобы сделать его более разборчивым. Приятное в этом сообщении об ошибках от сервера состоит в том, что оно перечисляет все причины, почему модуль был отклонен, а не только первую, с которой он столкнулся. Как вы можете видеть, модуль был отклонен по двум причинам: вы запросили два ЦП для контейнера, но максимальный лимит на ЦП для контейнера равен одному. Аналогичным образом этот модуль в целом запросил два ЦП, но максимум равен одному ЦП (если бы это был многоконтейнерный модуль, то даже если бы каждый отдельный контейнер запрашивал меньше максимального количества ЦП, то в сумме им все равно нужно было бы запросить меньше двух ЦП, для того чтобы передать максимум ЦП для модулей).

14.4.4 Применение стандартных ресурсных запросов и лимитов

Теперь давайте также посмотрим, как устанавливаются стандартные ресурсные запросы и лимиты для контейнеров, в которых они не задаются. Снова разверните модуль `kubia-manual` из главы 3:

```
$ kubectl create -f ../Chapter03/kubia-manual.yaml
pod "kubia-manual" created
```

До того, как вы настроили объект LimitRange, все модули были созданы без каких-либо ресурсных запросов или лимитов, но теперь при создании модуля

стандартные значения применяются автоматически. Это можно подтвердить, описав модуль `kubia-manual`, как показано в следующем ниже листинге.

Листинг 14.12. Инспектирование лимитов, автоматически применяемых к модулю

```
$ kubectl describe po kubia-manual
Name: kubia-manual
...
Containers:
  kubia:
    Limits:
      cpu: 200m
      memory: 100Mi
    Requests:
      cpu: 100m
      memory: 10Mi
```

Запросы и лимиты контейнера соответствуют указанным в объекте LimitRange. Если бы вы использовали другую спецификацию объекта LimitRange в другом пространстве имен, то модули, созданные в этом пространстве имен, очевидно, имели бы разные запросы и лимиты. Это позволяет администраторам настраивать стандартные, минимальные и максимальные ресурсы для модулей в расчете на пространство имен.

Если пространства имен используются для разделения различных групп разработчиков или для разделения этапа разработки, контроля качества и рабочего окружения, работающих в одном и том же кластере Kubernetes, использование разных объектов LimitRange в каждом пространстве имен гарантирует, что крупные модули могут создаваться только в определенных пространствах имен, в то время как другие ограничатся более мелкими модулями.

Но помните, что лимиты, сконфигурированные в объекте LimitRange, применяются только к каждому отдельному модулю/контейнеру. По-прежнему существует возможность создать много модулей и съесть все доступные в кластере ресурсы. Объекты LimitRange не обеспечивают от этого никакой защиты. Объект ResourceQuota, с другой стороны, это обеспечивает. Вы узнаете о них далее.

14.5 Лимитирование общего объема ресурсов, доступных в пространстве имен

Как вы убедились, объекты LimitRange применяются только к отдельным модулям. Однако администраторам кластера также нужен способ лимитировать общий объем ресурсов, доступных в пространстве имен. Это достигается путем создания объекта квоты ресурсов ResourceQuota.

14.5.1 Объект ResourceQuota

В главе 10 мы отметили, что несколько плагинов контроля допуска, работающих внутри сервера API, проверяют, может быть создан модуль или нет. В предыдущем разделе было сказано, что плагин LimitRanger применяет политики, сконфигурированные в ресурсах LimitRange. Точно так же плагин контроля допуска ResourceQuota проверяет, вызовет создаваемый модуль превышение сконфигурированной квоты ресурсов ResourceQuota или нет. Если это так, то создание модуля отклоняется. Поскольку квоты ресурсов применяются во время создания модуля, объект ResourceQuota влияет только на модули, созданные после создания объекта квоты ресурсов, и не влияет на существующие модули.

Объект ResourceQuota лимитирует объем вычислительных ресурсов, которые могут потребляться модулями, и объем хранилища, который могут потреблять заявки PersistentVolumeClaim в пространстве имен. Он также лимитирует количество модулей, заявок и других объектов API, которые пользователи могут создавать внутри пространства имен. Поскольку до сих пор вы в основном имели дело с процессором и памятью, давайте начнем с того, что укажем для них квоты.

Создание квоты ресурсов для ЦП и памяти

Как показано в следующем ниже листинге, общий объем ЦП и памяти, который могут использовать все модули в пространстве имен, определяется путем создания объекта ResourceQuota.

Листинг 14.13. Ресурс ResourceQuota для ЦП и памяти: quota-cpu-memory.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-and-mem
spec:
  hard:
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi
```

Вместо определения одного итогового значения для каждого ресурса можно определить отдельные итоговые значения для запросов и лимитов на ЦП и память. Вы заметите, что их структура немного отличается, по сравнению с описанием LimitRange. Здесь запросы и лимиты для всех ресурсов определяются в одном месте.

Данная квота ResourceQuota устанавливает максимальный объем ЦП, который модули в пространстве имен могут запросить, равный 400 миллиядрам. Максимальные общие лимиты на ЦП в пространстве имен равны 600 мил-

лиядрам. Для памяти максимальный общий объем запросов установлен в 200 MiB, тогда как лимиты установлены в 500 MiB.

Объект ResourceQuota применяется к пространству имен, в котором он создан, как и объект LimitRange, но он применяется ко всем ресурсным запросам и лимитам модулей в целом, а не к каждому отдельному модулю или контейнеру по отдельности, как показано на рис. 14.7.

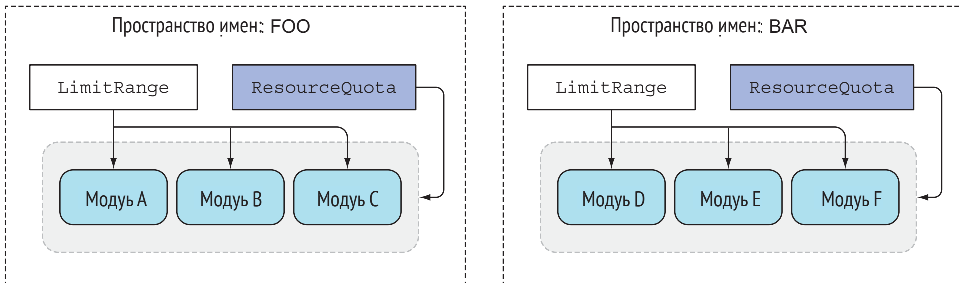


Рис. 14.7. Объекты LimitRange применяются к отдельным модулям; объекты ResourceQuota применяются ко всем модулям в пространстве имен

Инспектирование квот и их потребления

После того как вы отправите объект ResourceQuota на сервер API, вы можете применить команду `kubectl describe`, чтобы узнать, в каком объеме квота уже израсходована. Это показано в следующем ниже листинге.

Листинг 14.14. Инспектирование объекта ResourceQuota с помощью kubectl describe quota

```
$ kubectl describe quota
Name:           cpu-and-mem
Namespace:      default
Resource        Used   Hard
--------        ----   ----
limits.cpu      200m   600m
limits.memory   100Mi  500Mi
requests.cpu    100m   400m
requests.memory 10Mi   200Mi
```

У меня работает только модуль `kubia-manual`, поэтому столбец `Used` соответствует его ресурсным запросам и лимитам. Когда я запускаю дополнительные модули, их запросы и лимиты добавляются к используемым объемам.

Создание объекта LimitRange вместе с объектом ResourceQuota

Одна тонкость при создании объекта ResourceQuota заключается в том, что вы также захотите рядом с ним создать объект LimitRange. В вашем случае у вас сконфигурирован объект LimitRange из предыдущего раздела, но если у вас его не было, то вы не можете запустить модуль `kubia-manual`, потому что

он не указывает никаких ресурсных запросов или лимитов. Вот что произойдет в этом случае:

```
$ kubectl create -f ../Chapter03/kubia-manual.yaml
Error from server (Forbidden): error when creating "../Chapter03/kubia-
    manual.yaml": pods "kubia-manual" is forbidden: failed quota: cpu-andmem:
    must specify limits.cpu,limits.memory,requests.cpu,requests.memory
```

Когда квота для конкретного ресурса (ЦП или памяти) сконфигурирована (запрос или лимит), в модулях должен быть установлен запрос или лимит (соответственно) для того же ресурса; в противном случае модуль не будет принят сервером API. Вот почему наличие объекта LimitRange со стандартными для этих ресурсов настройками может немного упростить жизнь людям, которые занимаются созданием модулей.

14.5.2 Указание квоты для постоянного хранилища

Объект ResourceQuota также может лимитировать объем постоянного хранилища, которое может быть востребовано в пространстве имен. Это показано в следующем ниже листинге.

Листинг 14.15. Объект ResourceQuota для хранения: quota-storage.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage
spec:
  hard:
    requests.storage: 500Gi                                     <- Объем заявляемого хранилища в целом
    ssd.storageclass.storage.k8s.io/requests.storage: 300Gi     <- Объем памяти, заявляемый в хранилище класса ssd
    standard.storageclass.storage.k8s.io/requests.storage: 1Ti
```

В этом примере объем хранилища, который могут запросить все заявки PersistentVolumeClaim в пространстве имен, ограничен 500 GiB (записью `requests.storage` в объекте ResourceQuota). Но, как вы помните из главы 6, заявки PersistentVolumeClaim могут запрашивать динамически резервируемый постоянный том PersistentVolume определенного класса хранения StorageClass. Именно поэтому Kubernetes также позволяет индивидуально определять квоты для каждого класса хранения StorageClass. В предыдущем примере общий объем заявляемого хранилища SSD (обозначенного как StorageClass `ssd`) лимитирован 300 GiB. Менее производительное хранилище на жестких дисках HDD (StorageClass *standard*) лимитировно 1 TiB.

14.5.3 Лимитирование количества создаваемых объектов

Объект ResourceQuota также можно сконфигурировать для лимитирования количества модулей, контроллеров репликации, служб и других объектов

внутри одного пространства имен. Это позволяет администратору кластера ограничивать количество объектов, которые могут создаваться пользователями на основе их плана платежей, а также ограничивать количество общедоступных IP-адресов или портов узлов, которые могут использоваться службами.

В следующем ниже листинге показано, как может выглядеть объект ResourceQuota, ограничивающий количество объектов.

Листинг 14.16. Объект ResourceQuota для максимального количества ресурсов: quota-object-count.и yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: objects
spec:
  hard:
    pods: 10
    replicationcontrollers: 5
    secrets: 10
    configmaps: 10
    persistentvolumeclaims: 4
    services: 5
    services.loadbalancers: 1
    services.nodeports: 2
    ssd.storageclass.storage.k8s.io/persistentvolumeclaims: 2
```

В пространстве имен может быть создано только 10 модулей, 5 контроллеров репликации, 10 секретов, 10 словарей конфигурации и 4 заявки на получение постоянных томов

Можно создать пять служб, из которых не более одной может быть служба балансировки нагрузки и не более двух – службы NodePort

Только две заявки PVC могут запросить хранилище с классом хранения ssd

Объект ResourceQuota в этом списке позволяет пользователям создавать не более 10 модулей в пространстве имен, независимо от того, созданы они вручную или с помощью контроллера репликации ReplicationController, набора реплик ReplicaSet, набора демонов DaemonSet, задания Job и т. д. Он также устанавливает лимит на количество контроллеров репликации, равный пяти. Может быть создано максимум пять служб, из которых только одна может иметь тип `LoadBalancer` и только две могут быть службами `NodePort`. Подобно тому, как максимальный объем запрашиваемого хранилища может быть указан в расчете на класс хранения StorageClass, количество заявок PersistentVolumeClaim также может быть лимитировано в расчете на класс StorageClass.

В настоящее время квоты количества объектов можно задавать для следующих объектов:

- модули Pod;
- контроллеры репликации ReplicationController;
- секреты Secret;
- словари конфигурации ConfigMap;

- заявки PersistentVolumeClaim;
- службы Service (в общем случае) и для двух конкретных типов служб, таких как службы балансировки нагрузки LoadBalancer (services.loadbalancers) и службы NodePort (services.nodeports).

Наконец, можно даже установить квоту количества объектов для самих объектов ResourceQuota. Количество других объектов, таких как ReplicaSet, Job, Deployment, Ingress и т. д., пока не может быть лимитировано (но это, возможно, изменится к моменту публикации данной книги, поэтому, пожалуйста, проверьте документацию на наличие обновленной информации).

14.5.4 Указание квот для конкретных состояний модулей и/или классов QoS

Создававшиеся до сих пор квоты применялись ко всем модулям, независимо от их текущего состояния и класса QoS. Однако квоты также могут быть лимитированы набором *областей действия квот*. В настоящее время доступны четыре области действия: BestEffort, NotBestEffort, Terminating и NotTerminating.

Области действия BestEffort и NotBestEffort определяют, применяется ли квота к модулям с классом QoS BestEffort или с одним из двух других классов (то есть Burstable и Guaranteed).

Две другие области действия (Terminating и NotTerminating) не применяются к модулям, которые находятся (или не находятся) в процессе завершения работы, как можно заключить по их названию. Мы это еще не обсуждали, но вы можете указать, как долго каждый модуль может работать, прежде чем он будет завершен и помечен как Failed. Это делается путем установки поля activeDeadlineSeconds в спецификации модуля. Данное свойство определяет количество секунд, в течение которых модуль может быть активным на узле относительно его времени запуска, прежде чем он будет помечен как Failed, а затем завершен. Область действия квот Terminating применяется к модулям, у которых установлено поле activeDeadlineSeconds, в то время как область действия NotTerminating применяется к тем, у которых оно не установлено.

При создании объекта ResourceQuota можно указывать области действия, к которым он применяется. Для того чтобы квота применялась к модулю, он должен соответствовать всем указанным областям действия. Кроме того, то, что квота может лимитировать, зависит от области действия квоты. Область действия BestEffort может лимитировать только количество модулей, тогда как остальные три области действия могут лимитировать количество модулей, запросов на ЦП/память и лимитов на ЦП/память.

Если, например, требуется, чтобы квота применялась только к модулям Besteffort, NotTerminating, то вы можете создать объект ResourceQuota, показанный в следующем ниже листинге.

Листинг 14.17. Объект ResourceQuota для модуля BestEffort/NotTerminating: quota-scoped.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort-notterminating-pods
spec:
  scopes:
  - BestEffort
  - NotTerminating
  hard:
    pods: 4
```

Эта квота применяется только к модулям, которые имеют класс QoS BestEffort и у которых не установлен активный крайний срок

Могут существовать только четыре таких модуля

Эта квота гарантирует, что существует не более четырех модулей с классом QoS `BestEffort`, у которых нет активного крайнего срока. Если бы вместо этого квота была нацелена на модули `NotBestEffort`, то вы могли бы также указать поля `requests.cpu`, `requests.memory`, `limits.cpu` и `limits.memory`.

ПРИМЕЧАНИЕ. Прежде чем перейти к следующему разделу этой главы, пожалуйста, удалите все ресурсы ResourceQuota и LimitRange, которые вы создали. Они вам больше не понадобятся и могут помешать во время работы с примерами в последующих главах.

14.6 Мониторинг потребления ресурсов модуля

Правильная настройка ресурсных запросов и лимитов имеет решающее значение для получения максимальной отдачи от кластера Kubernetes. Если запросы заданы слишком высоко, то узлы кластера будут использоваться недостаточно, и вы будете тратить деньги впустую. Если же вы установите их слишком низко, то ваши приложения будут на голодном пайке или даже уничтожены убийцей OOM. Как найти «золотую середину» для запросов и лимитов?

Ее можно найти, отслеживая фактическое потребление ресурсов вашими контейнерами в соответствии с ожидаемыми уровнями загрузки. После того как к приложению будет открыт публичный доступ, вы должны продолжать за ним следить и корректировать ресурсные запросы и лимиты, если это необходимо.

14.6.1 Сбор и извлечение фактических данных потребления ресурсов

Каким образом можно отслеживать приложения, работающие в Kubernetes? К счастью, сам Kubelet уже содержит агент cAdvisor, который собирает базовый набор данных о потреблении ресурсов как для отдельных контейнеров, работающих на узле, так и для узла в целом. Для централизованного сбора

этих статистических данных по всему кластеру необходимо запустить дополнительный компонент под названием Heapster.

Heapster работает как модуль на одном из узлов и предоставляется через обычную службу Kubernetes, которая делает его доступным на стабильном IP-адресе. Он собирает данные от всех агентов cAdvisor в кластере и предоставляет их в одном месте. На рис. 14.8 показан поток метрических данных из модулей через агента cAdvisor и, наконец, в агрегаторе Heapster.

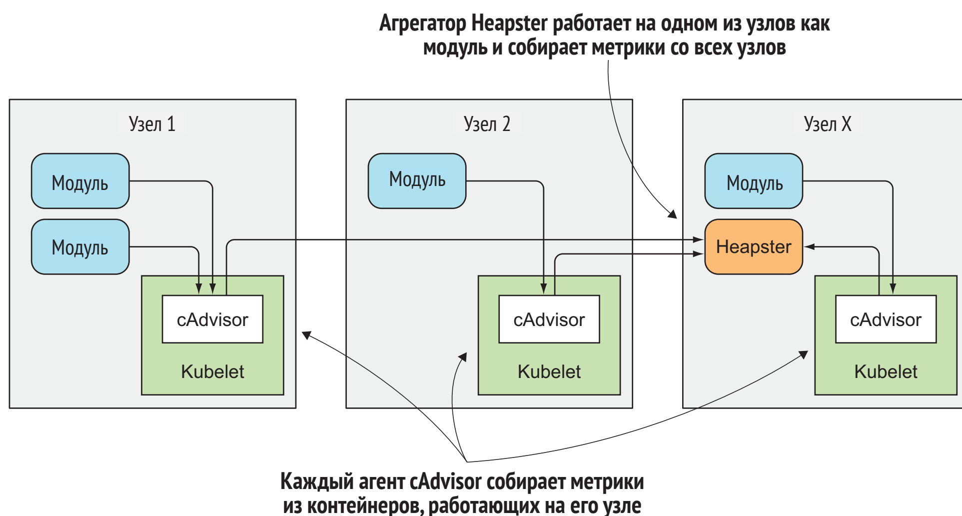


Рис. 14.8. Поток метрических данных в агрегаторе Heapster

Стрелки на рисунке показывают потоки метрических данных. Они не показывают, какой компонент подключается к какому из них для получения данных. Модули (или контейнеры, работающие в них) ничего не знают об агенте cAdvisor, а агент cAdvisor ничего не знает об агрегаторе Heapster. Именно Heapster подключается ко всем агентам cAdvisor, и именно агенты cAdvisor собирают данные об использовании контейнера и узла без необходимости обмениваться с процессами, работающими внутри контейнеров модулей.

Активация Heapster

Если вы используете кластер в Google Kubernetes Engine, то агрегатор Heapster активирован по умолчанию. Если же вы используете Minikube, то он доступен как надстройка и может быть активирован с помощью следующей ниже команды:

```
$ minikube addons enable heapster
heapster was successfully enabled
```

По поводу запуска агерегатора Heapster вручную в других типах кластеров Kubernetes вы можете обратиться к инструкциям, расположенным по адресу https://github.com/kubernetes/heapster.

После активирования агрегатора Heapster необходимо подождать несколько минут, пока он соберет метрики, прежде чем вы увидите статистику потребления ресурсов для вашего кластера, поэтому будьте терпеливы.

Вывод информации о потреблении ЦП и памяти для узлов кластера

Запуск агрегатора Heapster в кластере позволяет получать информацию о потреблении ресурсов для узлов и отдельных модулей с помощью команды `kubectl top`. Для того чтобы узнать, сколько ЦП и памяти используется на узлах, вы можете выполнить команду, показанную в следующем ниже листинге.

Листинг 14.18. Фактическое потребление ЦП и памяти узлами

```
$ kubectl top node
NAME       CPU(cores)  CPU%  MEMORY(bytes)  MEMORY%
minikube   170m        8%    556Mi          27%
```

Этот результат показывает фактическое, текущее потребление ЦП и памяти всеми модулями, работающими на узле, в отличие от команды `kubectl describe node`, которая показывает объем ЦП и памяти в запросах и лимитах вместо фактических данных о потреблении времени выполнения.

Вывод информации о потреблении ЦП и памяти для отдельных модулей

Для того чтобы узнать, сколько потребляет каждый отдельный модуль, вы можете применить команду `kubectl top pod`, как показано в следующем ниже листинге.

Листинг 14.19. Фактическое потребление процессоров и памяти в модулях

```
$ kubectl top pod --all-namespaces
NAMESPACE     NAME                             CPU(cores)   MEMORY(bytes)
kube-system   influxdb-grafana-2r2w9           1m           32Mi
kube-system   heapster-40j6d                   0m           18Mi
default       kubia-3773182134-63bmb           0m           9Mi
kube-system   kube-dns-v20-z0hq6               1m           11Mi
kube-system   kubernetes-dashboard-r53mc       0m           14Mi
kube-system   kube-addon-manager-minikube      7m           33Mi
```

Результаты обеих этих команд довольно просты, поэтому вам, вероятно, не нужно, чтобы я их объяснял, но мне нужно предупредить вас об одной вещи. Иногда команда `top pod` отказывается показывать какие-либо метрики и вместо этого выводит ошибку, как вот здесь:

```
$ kubectl top pod
W0312 22:12:58.021885 15126 top_pod.go:186] Metrics not available for pod
    default/kubia-3773182134-63bmb, age: 1h24m19.021873823s
```

```
error: Metrics not available for pod default/kubia-3773182134-63bmb, age:
    1h24m19.021873823s
```

Если это произойдет, не начинайте искать причину ошибки. Расслабьтесь, подождите некоторое время и повторите команду – это может занять несколько минут, но в конечном итоге метрики должны появиться. Команда `kubectl top` получает метрики из Heapster, который собирает данные за несколько минут и не предоставляет их немедленно.

> **СОВЕТ.** Для того чтобы просмотреть информацию о потреблении ресурсов в отдельных контейнерах вместо модулей, вы можете использовать параметр `--containers`.

14.6.2 Хранение и анализ исторической статистики потребления ресурсов

Команда `top` показывает только текущее потребление ресурсов – к примеру, она не показывает объем процессора или памяти, который был потреблен вашими модулями в течение последнего часа, за вчерашний день или за предыдущую неделю. Фактически агент cAdvisor и агрегатор Heapster хранят данные о потреблении ресурсов только в течение короткого промежутка времени. Если вы хотите анализировать потребление ресурсов модулями в течение более длительных периодов времени, то вам нужно запустить дополнительные инструменты.

При использовании Google Kubernetes Engine вы можете отслеживать работу вашего кластера с помощью мониторинга Google Cloud Monitoring, но когда вы используете свой собственный локальный кластер Kubernetes (посредством Minikube или иным способом), то обычно для хранения статистики используют InfluxDB, для визуализации и анализа – Grafana.

Знакомство с базой данных InfluxDB и пакетом Grafana

InfluxDB – это база данных временных рядов с открытым исходным кодом, которая идеально подходит для хранения метрик приложений и других данных мониторинга. Grafana, также с открытым исходным кодом, представляет собой пакет аналитики и визуализации с красивой веб-консолью, которая позволяет визуализировать данные, хранящиеся в базе данных InfluxDB, и узнавать, как использование ресурсов вашего приложения ведет себя с течением времени (пример, демонстрирующий три диаграммы пакета Grafana, показан на рис. 14.9).

Запуск базы данных InfluxDB и пакета Grafana в кластере

И база данных InfluxDB, и пакет Grafana могут работать как модули. Их развертывание не представляет труда. Все необходимые манифесты доступны в репозитории Git агрегатора Heapster на http://github.com/kubernetes/heapster/tree/master/deploy/kube-config/influxdb.

При использовании Minikube их даже не нужно развертывать вручную, так как они развертываются вместе с агрегатором Heapster при активировании надстройки агрегатора Heapster.

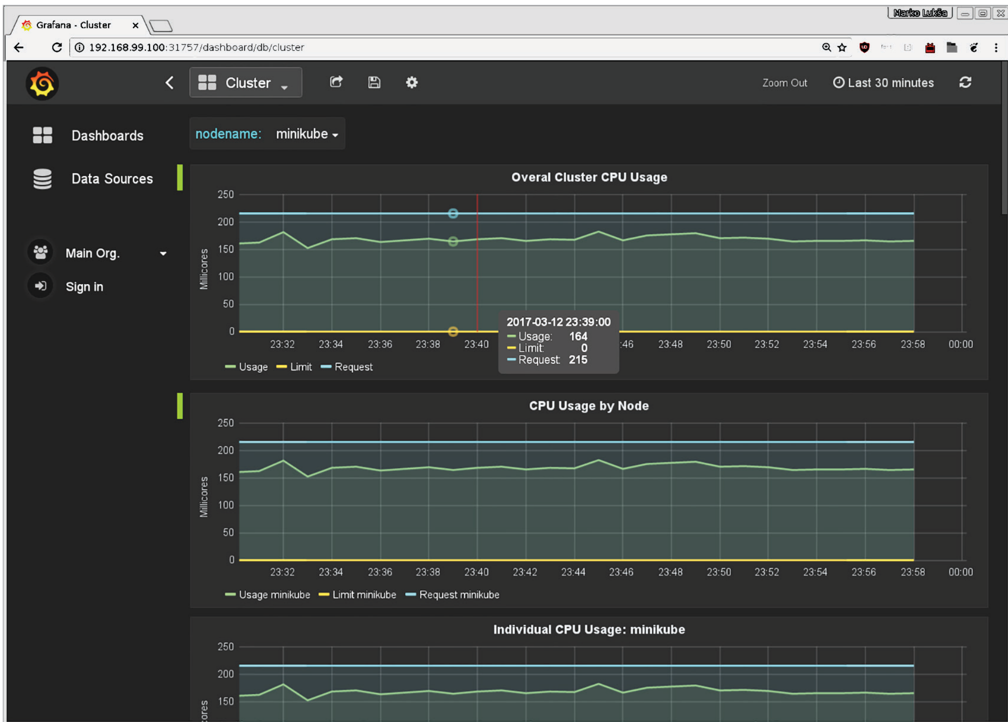


Рис. 14.9. Панель мониторинга Grafana, показывающая потребление ЦП в кластере

Анализ потребления ресурсов с помощью Grafana

Для того чтобы узнать, сколько ресурсов требуется вашему модулю с течением времени, откройте веб-консоль Grafana и изучите предопределенные панели мониторинга. Как правило, вы можете выяснить URL-адрес веб-консоли Grafana с помощью команды `kubectl cluster-info`:

```
$ kubectl cluster-info
...
monitoring-grafana is running at
    https://192.168.99.100:8443/api/v1/proxy/namespaces/kube-
    system/services/monitoring-grafana
```

При использовании Minikube веб-консоль Grafana предоставляется через службу `NodePort`, поэтому ее можно открыть в браузере с помощью следующей ниже команды:

```
$ minikube service monitoring-grafana -n kube-system
```

```
Opening kubernetes service kube-system/monitoring-grafana in default
    browser...
```

В результате откроется новое окно браузера или вкладка с начальным экраном Grafana. Справа вы увидите список панелей мониторинга, содержащий две записи:

- Cluster;
- Pods.

Для того чтобы просмотреть статистику потребления ресурсов узлами, откройте панель мониторинга Cluster. Там вы увидите несколько диаграмм, показывающих общее потребление кластера, потребление по узлам и индивидуальное потребление ЦП, памяти, сети и файловой системы. На диаграммах будет показано не только фактическое потребление, но и запросы и лимиты для этих ресурсов (где они применяются).

Если вы затем переключитесь на панель мониторинга Pods, то сможете проверить потребление ресурсов по каждому отдельному модулю, при этом снова будут показаны запросы и лимиты вместе с фактическим потреблением.

Исходно на диаграммах отображается статистика за последние 30 минут, но вы можете уменьшить масштаб и просматривать данные за гораздо более длительные периоды времени: дни, месяцы или даже годы.

Использование информации, выводимой на диаграммах

Взглянув на диаграммы, вы можете быстро увидеть, нужно ли повысить ресурсные запросы или лимиты, которые вы установили для ваших модулей, или же они могут быть снижены, чтобы дать большему количеству модулей поместиться на ваших узлах. Давайте рассмотрим пример. На рис. 14.10 показаны диаграммы ЦП и памяти для модуля.

В крайнем правом углу верхней диаграммы можно увидеть, что модуль использует ЦП больше, чем было запрошено в манифесте модуля. Хотя это не проблематично, когда речь идет о единственном работающем на узле модуле, следует иметь в виду, что модулю гарантируется только такой объем ресурса, который был запрошен посредством ресурсных запросов. Ваш модуль может работать нормально, но, когда на том же узле развертываются другие модули и начинают использовать ЦП, процессорное время модуля может быть лимитировано. По этой причине, для того чтобы модуль гарантированно мог использовать столько ЦП, сколько ему нужно в любое время, вам следует повысить ресурсный запрос на ЦП для контейнера модуля.

Нижняя диаграмма показывает потребление памяти и запрос модуля. Здесь ситуация прямо противоположная. Объем потребляемой модулем памяти намного ниже, что было запрошено в секции `spec` модуля. Запрошенная память зарезервирована за модулем и не будет доступна для других модулей. Следовательно, неиспользуемая память расходуется впустую. Вам следует уменьшить запрос на память модулем, чтобы сделать память доступной для других работающих на узле модулей.

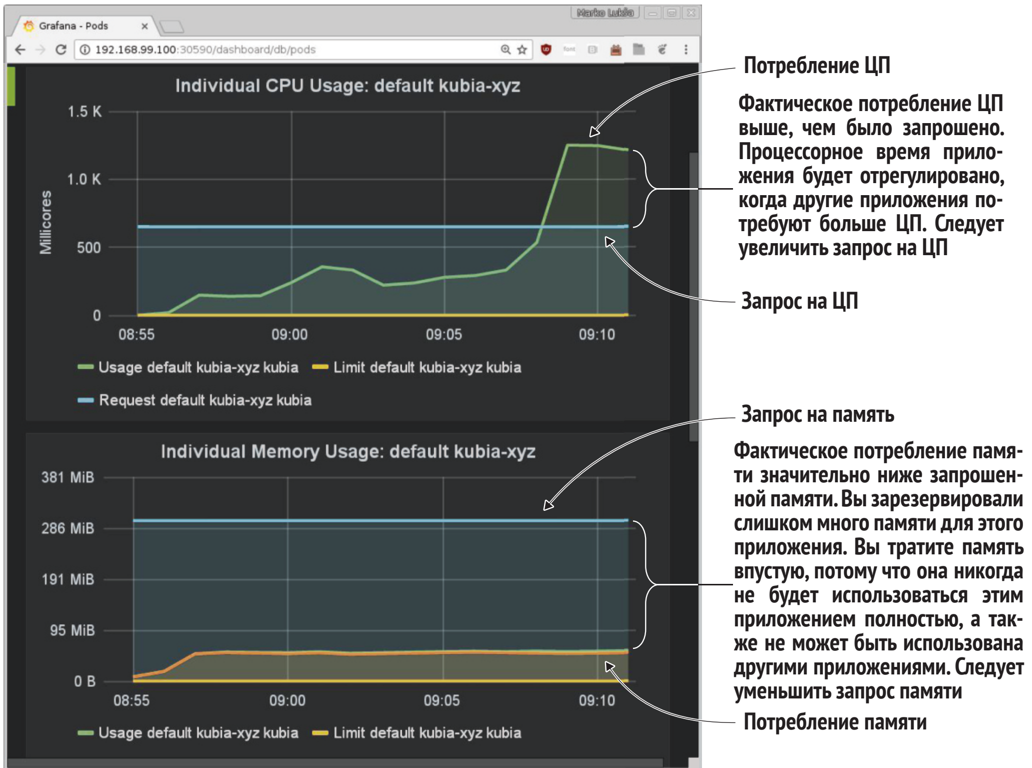


Рис. 14.10. Диаграмма потребления модулем ЦП и памяти

14.7 Резюме

В этой главе было показано, что, для того чтобы все работало гладко, вам следует учитывать потребление ресурсов модулем и конфигурировать ресурсные запросы и лимиты для модуля. Основные выводы из этой главы следующие:

- указание ресурсных запросов помогает Kubernetes планировать работу модулей в кластере;
- указание лимитов на ресурсы не позволяет модулям держать на голодном ресурсном пайке другие модули;
- неиспользованное процессорное время выделяется системой Kubernetes на основе запросов ЦП контейнерами;
- контейнеры никогда не уничтожаются, если они пытаются использовать слишком много ЦП, но они уничтожаются, если они пытаются использовать слишком много памяти;
- в перегруженной системе контейнеры также уничтожаются, освобождая память для более важных модулей, на основе классов QoS модулей и фактического потребления памяти;

- объекты LimitRange можно использовать для определения минимальных, максимальных и стандартных ресурсных запросов и лимитов для отдельных модулей;
- объекты ResourceQuota можно использовать для лимитирования объема ресурсов, доступных для всех модулей в пространстве имен;
- для того чтобы узнать, насколько высоки ресурсные запросы и лимиты модуля, вам нужно следить за тем, как модуль потребляет ресурсы в течение достаточно длительного периода времени.

В следующей главе вы увидите, как эти метрики могут использоваться системой Kubernetes для автоматического масштабирования модулей.

Глава 15

Автоматическое масштабирование модулей и узлов кластера

Эта глава посвящена:

- настройке автоматического горизонтального масштабирования модулей на основе задействованности ЦП;
- настройке автоматического горизонтального масштабирования модулей на основе собственных настраиваемых метрик;
- объяснению причины, почему вертикальное масштабирование модулей пока невозможно;
- автоматическому горизонтальному масштабированию узлов кластера.

Приложения, работающие в модулях, можно масштабировать вручную, увеличивая поле `replicas` в контроллере репликации ReplicationController, наборе реплик ReplicaSet, развертывании Deployment или другом масштабируемом ресурсе. Модули также можно масштабировать по вертикали, увеличивая ресурсные запросы и лимиты их контейнеров (хотя в настоящее время это можно делать только во время создания модуля, а не во время работы модуля). Ручное масштабирование подходит для случаев, когда вы можете заранее предвидеть скачки нагрузки или когда нагрузка постепенно изменяется в течение длительных периодов времени. Вместе с тем вариант, когда для обработки внезапного, непредсказуемого увеличения трафика требуется ручное вмешательство, не самый лучший.

К счастью, система Kubernetes может отслеживать ваши модули и масштабировать их автоматически, как только она обнаруживает увеличение в потреблении ЦП либо какого-либо другого метрического показателя. Работая на

облачной инфраструктуре, она может даже разворачивать дополнительные узлы, если существующие больше не могут принимать модули. В этой главе объясняется, как сделать так, чтобы Kubernetes выполняла автомасштабирование как модуля, так и узла.

Функционал автомасштабирования в Kubernetes был полностью переписан между версиями 1.6 и 1.7, поэтому имейте в виду, что в интернете вы можете найти устаревшую информацию по этому вопросу.

15.1 Горизонтальное автомасштабирование модуля

Горизонтальное автомасштабирование модуля – это автоматическое масштабирование количества реплик модуля, управляемых контроллером. Оно выполняется горизонтальным контроллером, который активируется и конфигурируется путем создания ресурса HorizontalPodAutoscaler (HPA). Данный контроллер периодически проверяет метрики модуля, вычисляет количество реплик, необходимое для соответствия целевому значению метрики, сконфигурированной в ресурсе HorizontalPodAutoscaler, и настраивает поле `replicas` на целевом ресурсе (развертывании Deployment, наборе реплик ReplicaSet, контроллере репликации ReplicationController или наборе модулей с внутренним состоянием StatefulSet).

15.1.1 Процесс автомасштабирования

Процесс автомасштабирования можно разделить на три этапа:

- получение метрик всех модулей, управляемых масштабируемым ресурсным объектом;
- рассчет количества модулей, необходимого для приведения метрик к указанному целевому значению (или близкому к нему);
- обновление поля `replicas` масштабируемого ресурса.

Далее мы рассмотрим все три этапа.

Получение метрик модуля

Автопреобразователь масштаба сам не выполняет сбор метрик модуля. Он получает метрики из другого источника. Как мы видели в предыдущей главе, метрики модуля и узла собираются агентом под названием cAdvisor, который выполняется в Kubelet на каждом узле, а затем агрегируется кластерным компонентом под названием Heapster. Контроллер автопреобразователя горизонтального масштаба модуля получает метрики всех модулей, запрашивая агрегатор Heapster посредством вызовов REST. Поток метрических данных показан на рис. 15.1 (правда, все связи инициируются в противоположном направлении).

Из этого вытекает, что, для того чтобы автомасштабирование сработало, агрегатор Heapster должен быть запущен в кластере. Если вы используете

Minikube и следили за изложением в предыдущей главе, то агрегатор Heapster уже должен быть активирован в вашем кластере. Если нет, то, прежде чем приступать к опробованию примеров автомасштабирования, обязательно активируйте надстройку агрегатора Heapster.

Хотя вам не нужно опрашивать агрегатор Heapster напрямую, однако если вы в этом заинтересованы, то вы найдете и модуль Heapster, и службу, через которую к нему предоставляется доступ, в пространстве имен `kube-system`.

Рис. 15.1. Поток метрик из модуля (модулей) в автопреобразователь(и) горизонтального масштаба

Обзор изменений, связанных с тем, как автопреобразователь масштаба получает метрики

До Kubernetes версии 1.6 автопреобразователь горизонтального масштаба модуля HorizontalPodAutoscaler получал метрики от агрегатора Heapster напрямую. В версии 1.8 автопреобразователь масштаба может получать метрики через агрегированную версию API ресурсных метрик, запустив менеджер контроллеров (Controller Manager) с флагом `--horizontal-pod-autoscaler-use-rest-clients=true`. Начиная с версии 1.9 это поведение будет активировано по умолчанию.

Основной сервер API сам не предоставляет доступа к метрикам. Начиная с версии 1.7 Kubernetes предоставляет возможность регистрировать несколько серверов API и отображать их как один сервер API. Это позволяет ему обеспечивать доступ к метрикам через один из этих внутренних серверов API. Мы объясним агрегацию серверов API в последней главе.

Выбор сборщика метрик для использования в их кластерах зависит от администраторов кластера. Обычно для предоставления доступа к метрикам в соответствующих путях API и в соответствующем формате требуется простой слой трансляции.

Расчет необходимого количества модулей

После того как автопреобразователь масштаба получил метрики для всех модулей, принадлежащих ресурсу, который автопреобразователь масштабирует Deployment, ReplicaSet, ReplicationController или StatefulSet, он может использовать эти метрики, чтобы выяснить необходимое количество реплик. Он должен найти число, которое сделает среднее значение метрики для всех реплик как можно ближе к сконфигурированному целевому значению. Входными данными для этого вычисления является метрика модуля (возможно,

несколько метрик на модуль), а выходными данными является целое число (количество реплик модуля).

Когда автопреобразователь масштаба сконфигурирован на рассмотрение только одной метрики, расчет необходимого количества реплик не представляет труда. Требуется лишь просуммировать значения метрик всех модулей, разделив их на целевое значение, установленное на ресурсе HorizontalPodAutoscaler, а затем округлить до ближайшего целого. Фактическая процедура вычисления немного сложнее, чем тут, потому что она также проверяет, чтобы автопреобразователь масштаба не метался туда-сюда, когда значение метрики неустойчиво и быстро меняется.

Когда автомасштабирование основано на нескольких метриках модуля (например, потреблении ЦП и запросах в секунду [QPS]), вычисление ненамного сложнее. Автопреобразователь масштаба вычисляет количество реплик для каждой метрики по отдельности, а затем принимает наибольшее значение (например, если для достижения целевого потребления ЦП требуется четыре модуля, а для достижения целевого QPS требуется три модуля, то автопреобразователь промасштабирует до четырех модулей). Рисунок 15.2 показывает этот пример.

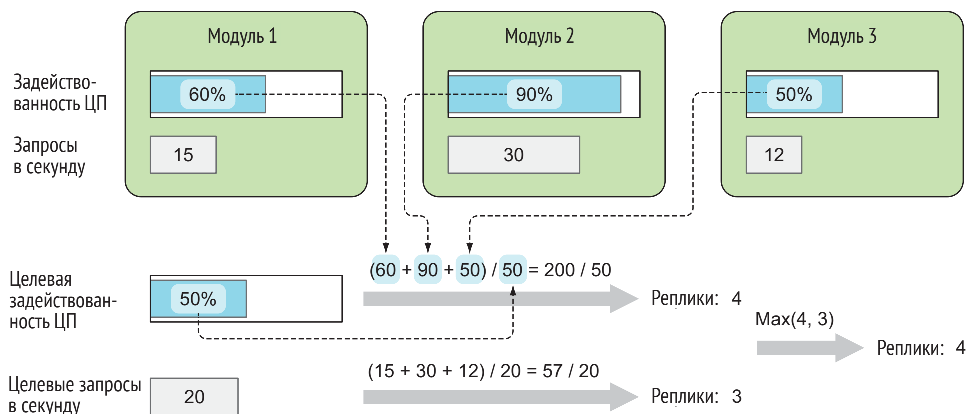


Рис. 15.2. Расчет количества реплик из двух метрик

Обновление требуемого количества реплик для масштабируемого ресурса

Последним этапом операции автомасштабирования является обновление поля требуемого количества реплик для объекта масштабируемого ресурса (например, набора реплик ReplicaSet), а затем предоставление контроллеру набора реплик возможности развернуть дополнительные модули или удалить лишние.

Контроллер автопреобразования масштаба изменяет поле `replicas` масштабируемого ресурса с помощью подресурса Scale. Он дает возможность автопреобразователю масштаба делать свою работу, не зная никаких подробностей о ресурсе, который он масштабирует, за исключением того, к чему предоставляется доступ посредством подресурса Scale (см. рис. 15.3).

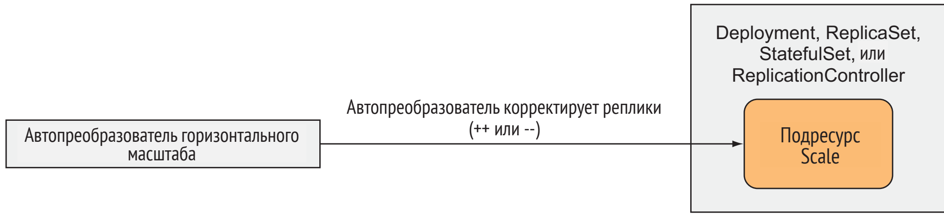


Рис. 15.3. Автопреобразователь горизонтального масштаба модуля работает, внося изменения, только на подресурсе Scale

Это позволяет автопреобразователю масштаба оперировать на любом масштабируемом ресурсе, в случае если сервер API предоставляет доступ к подресурсу Scale. В настоящее время он доступен для:

- развертываний Deployment;
- наборов реплик ReplicaSet;
- контроллеров репликации ReplicationController;
- наборов модулей с внутренним состоянием StatefulSet.

В настоящее время это единственные объекты, которые можно прикрепить к автопреобразователю масштаба.

Полный процесс автомасштабирования

Теперь, когда вы разбираетесь в трех этапах автомасштабирования, давайте визуализируем все компоненты, участвующие в данном процессе автомасштабирования. Они показаны на рис. 15.4.

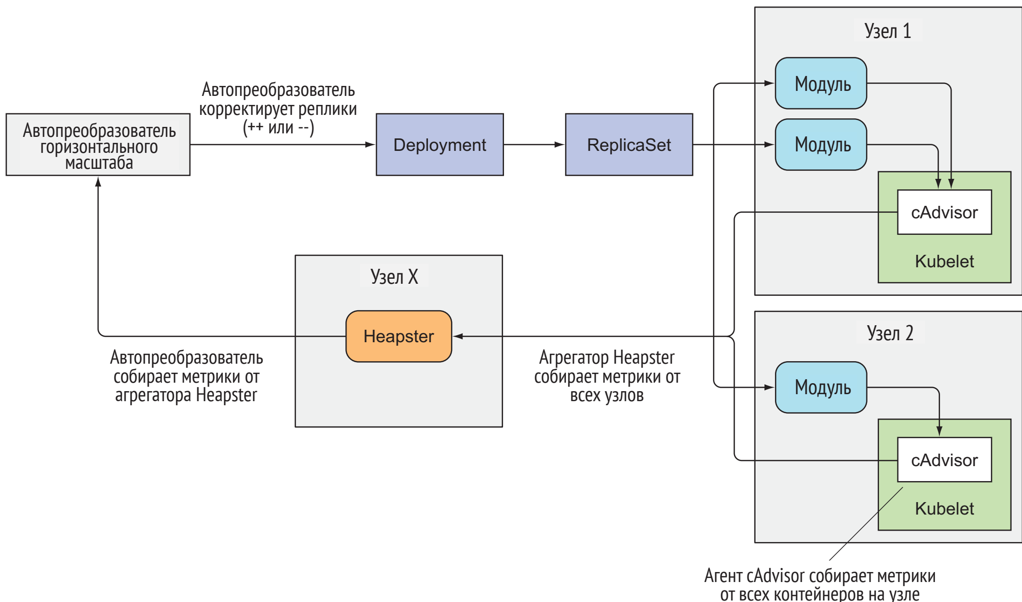


Рис. 15.4. Как автопреобразователь масштаба получает метрики и масштабирует целевое развертывание

Стрелки, ведущие от модулей к агентам cAdvisor, которые продолжаются вплоть до агрегатора Heapster и, наконец, до автопреобразователя горизонтального масштаба модулей, указывают направление потока метрических данных. Важно знать, что каждый компонент периодически получает метрики от других компонентов (то есть агент cAdvisor получает метрики от модулей в бесконечном цикле; то же самое верно и для агрегатора Heapster, и для контроллера HPA). Конечный эффект заключается в том, что для распространения данных метрик и выполнения операции масштабирования требуется довольно много времени. Это происходит не сразу. Имейте это в виду, когда вы далее будете наблюдать за автопреобразователем масштаба в действии.

15.1.2 Масштабирование на основе задействованности ЦП

Возможно, наиболее важной метрикой, на которой вы хотите основывать автомасштабирование, является объем ЦП, потребляемый процессами, работающими внутри ваших модулей. Представьте, что у вас есть несколько модулей, которые предоставляют службу. Когда их потребление ЦП достигает 100%, очевидно, что они больше не смогут справляться с нагрузкой и должны быть промасштабированы либо вверх (вертикальное масштабирование – увеличение объема ЦП, который модули могут использовать), либо вширь (горизонтальное масштабирование – увеличение количества модулей). Поскольку мы здесь говорим об автопреобразователе горизонтального масштаба модулей, то фокусируемся только на масштабировании вширь (увеличении количества модулей). В результате него среднее потребление ЦП должно снизиться.

Поскольку потребление ЦП обычно нестабильно, имеет смысл выполнять масштабирование даже до того, как ЦП будет загружен по завязку – возможно, когда средняя загрузка ЦП по всем модулям достигает или превышает 80%. Но 80% *чего* именно?

> **СОВЕТ.** Всегда устанавливайте целевое потребление ЦП значительно ниже 100% (и определенно никогда выше 90%), чтобы оставить достаточно места для корректировки неожиданных всплесков в нагрузке.

Как вы помните из предыдущей главы, процессу, работающему внутри контейнера, гарантирован объем ЦП, запрашиваемый посредством ресурсных запросов, задаваемых для контейнера. Но иногда, когда никакой другой процесс не нуждается в ЦП, этот процесс может использовать весь доступный ЦП на узле. Когда кто-то говорит, что модуль потребляет 80% ЦП, не совсем ясно, что он имеет в виду: 80% ЦП узла, 80% гарантированного ЦП модуля (ресурсного запроса) или 80% жесткого лимита, настроенного для модуля посредством ресурсных лимитов.

Что касается автопреобразователя масштаба, то при определении задействованности ЦП модуля важен только гарантированный объем ЦП модуля (запросы на ЦП). Автопреобразователь масштаба сравнивает фактическое потребление ЦП модулем и его запросы на ЦП, имея в виду, что, для того чтобы

автопреобразователь масштаба определил процент задействованности ЦП, в модулях, которые вы автомасштабируете, должны быть установлены запросы на ЦП (прямо или косвенно через объект LimitRange).

Создание автопреобразователя горизонтального масштаба модулей на основе потребления ЦП

Давайте сейчас посмотрим, как создать автопреобразователь HorizontalPodAutoscaler и сконфигурировать его для масштабирования модулей на основе задействованности их ЦП. Вы создадите развертывание, аналогичное тому, которое описано в главе 9, но, как мы уже обсуждали, вам нужно убедиться, что все модули, созданные развертыванием, имеют ресурсные запросы на ЦП, которые указываются для того, чтобы обеспечить возможность автомасштабирования. Вам нужно добавить ресурсный запрос на ЦП в шаблон модуля развертывания, как показано в следующем ниже листинге.

Листинг 15.1. Развертывание с установленным запросом на ЦП: deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3          <-- Ручное присвоение (начального) требуемого
  template:                количества реплик, равное трем
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:                       Запуск образа kubia:v1
      - image: luksa/kubia:v1      <--
        name: nodejs
        resources:               Запрос 100 миллиядер
          requests:         <--  ЦП на модуль
            cpu: 100m
```

Это обычный объект развертывания – он еще не использует автомасштабирования. Он будет запускать три экземпляра приложения NodeJS `kubia`, при этом каждый экземпляр запрашивает 100 миллиядер ЦП.

После создания развертывания, для того чтобы активировать горизонтальное автомасштабирование его модулей, необходимо создать объект HorizontalPodAutoscaler (HPA) и указать его на развертывание. Вы могли бы подготовить для HPA манифест YAML и отправить его на сервер, но существует более простой способ – с помощью команды `kubectl autoscale`:

```
$ kubectl autoscale deployment kubia --cpu-percent=30 --min=1 --max=5
deployment "kubia" autoscaled
```

Эта команда создает для вас объект HPA и назначает развертывание `kubia` в качестве цели масштабирования. Вы устанавливаете целевую задействованность ЦП для модулей в размере 30% и указываете минимальное и максимальное количество реплик. Автопреобразователь масштаба будет постоянно корректировать количество реплик, чтобы удерживать их задействованность ЦП на уровне 30%, но он никогда не уменьшит масштаб до менее чем одной или не увеличит масштаб до более чем пяти реплик.

> **СОВЕТ.** Всегда старайтесь автомасштабировать развертывания, а не лежащие в их основе наборы реплик ReplicaSet. Благодаря этому вы гарантируете, что требуемое количество реплик сохранится во всех обновлениях приложения (напомним, что развертывание создает новый набор реплик для каждой версии). То же правило применимо и к ручному масштабированию.

Давайте рассмотрим определение ресурса HorizontalPodAutoscaler, чтобы разобраться в нем получше. Это показано в следующем ниже листинге.

Листинг 15.2. Определение YAML ресурса HorizontalPodAutoscaler

```
$ kubectl get hpa.v2beta1.autoscaling kubia -o yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: kubia
  ...
spec:
  maxReplicas: 5
  metrics:
  - resource:
      name: cpu
      targetAverageUtilization: 30
    type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: kubia
status:
  currentMetrics: []
  currentReplicas: 3
  desiredReplicas: 0
```

Ресурсы HPA находятся в группе API autoscaling

У каждого HPA есть имя (оно не обязательно должно совпадать с именем развертывания, как в этом случае)

Минимальное и максимальное количество реплик, которые вы указали

Вы хотите, чтобы автопреобразователь масштаба скорректировал количество модулей, для того чтобы каждый из них использовал 30% запрошенного ЦП

Целевой ресурс, на котором этот автопреобразователь масштаба будет действовать

Текущий статус автопреобразователя масштаба

ПРИМЕЧАНИЕ. Существует несколько версий ресурсов HPA: новая `autoscaling/v2beta1` и старая `autoscaling/v1`. Здесь вы запрашиваете новую версию.

Просмотр первого события автоматического масштабирования

Прежде чем автопреобразователь масштаба сможет принять меры, требуется некоторое время, для того чтобы агент cAdvisor получил метрики ЦП и агрегатор Heapster их собрал. Если в течение этого промежутка времени вы выведете на экран ресурс HPA с помощью команды `kubectl get`, то столбец `TARGETS` покажет `<unknown>`:

```
$ kubectl get hpa
NAME   REFERENCE         TARGETS         MINPODS  MAXPODS  REPLICAS
kubia  Deployment/kubia  <unknown> / 30%  1        5        0
```

Поскольку вы работаете с тремя модулями, которые в настоящее время не получают запросов, а значит, их загрузка ЦП должна быть близка к нулю, вы должны ожидать, что автопреобразователь масштаба промасштабирует их вниз до одного модуля, потому что даже с одним модулем задействованность ЦП все равно будет ниже целевого значения 30%.

И безусловно, автопреобразователь масштаба сделает именно это. Вскоре он промасштабирует развертывание до одной реплики:

```
$ kubectl get deployment
NAME   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
kubia  1        1        1           1          23m
```

Помните, что автопреобразователь масштаба корректирует количество реплик только на развертывании. Затем контроллер развертывания примется за обновление необходимого количества реплик на объекте ReplicaSet, который потом заставляет контроллер набора ReplicaSet удалить два лишних модуля, оставив работающим один модуль.

Как показано в следующем ниже листинге, можно применить команду `kubectl describe`, чтобы увидеть дополнительную информацию об объекте HorizontalPodAutoscaler и работе лежащего в его основе контроллера.

Листинг 15.3. Инспектирование объекта HorizontalPodAutoscaler с помощью команды kubectl describe

```
$ kubectl describe hpa
Name:                         kubia
Namespace:                    default
Labels:                       <none>
Annotations:                  <none>
CreationTimestamp:            Sat, 03 Jun 2017 12:59:57 +0200
Reference:                    Deployment/kubia
```

```
Metrics:                         ( current / target )
  resource cpu on pods
  (as a percentage of request):  0% (0) / 30%
Min replicas:                    1
Max replicas:                    5
Events:
From                       Reason             Message
----                       ------             ---
horizontal-pod-autoscaler  SuccessfulRescale  New size: 1; reason: All
                                              metrics below target
```

> **ПРИМЕЧАНИЕ.** Полученный результат был видоизменен, чтобы сделать его более читаемым.

Обратите внимание на таблицу событий в нижней части листинга. Вы видите, что автопреобразователь горизонтального масштаба модулей успешно уменьшил масштаб до одной реплики, потому что все метрики были ниже целевых.

Инициирование увеличения масштаба

Вы уже стали свидетелями первого события автоматического масштабирования (уменьшения масштаба). Теперь вы начнете отправлять запросы на свой модуль, тем самым увеличивая его потребление ЦП, и вы должны увидеть, что автопреобразователь масштаба обнаружит это и запустит дополнительные модули.

Для того чтобы вы могли попадать в модули через единый URL, вам нужно предоставить к ним доступ через службу. Вы, наверное, помните, что самый простой способ для этого – применить команду `kubectl expose`:

```
$ kubectl expose deployment kubia --port=80 --target-port=8080
service "kubia" exposed
```

Перед тем как начать нагружать свой модуль(и) запросами, вы, возможно, захотите выполнить следующую ниже команду в отдельном терминале, чтобы следить за тем, что происходит с объектом HorizontalPodAutoscaler и развертыванием. Эта команда и ее результат показаны в следующем ниже листинге.

Листинг 15.4. Параллельное наблюдение за несколькими ресурсами

```
$ watch -n 1 kubectl get hpa,deployment
Every 1.0s: kubectl get hpa,deployment
NAME       REFERENCE         TARGETS    MINPODS  MAXPODS  REPLICAS  AGE
hpa/kubia  Deployment/kubia  0% / 30%   1        5        1         45m
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
deploy/kubia  1        1        1           1          56m
```

СОВЕТ. Список типов ресурсов можно выводить с помощью команды `kubectl get`, разделяя их запятой.

Если вы используете OSX, то вам потребуется заменить команду `watch` на цикл, чтобы вручную периодически выполнять команду `kubectl get`, либо использовать параметр `--watch` инструмента `kubectl`. Но хотя простая команда `kubectl get` может одновременно выводить несколько типов ресурсов, это не относится к случаю, когда используется вышеупомянутый параметр `--watch`, поэтому, если вы хотите наблюдать и за HPA, и за объектами развертывания, то вам нужно будет использовать два терминала.

Во время работы модуля, генерирующего нагрузку, следите за состоянием этих двух объектов. В другом терминале выполните следующую ниже команду:

```
$ kubectl run -it --rm --restart=Never loadgenerator --image=busybox
➥   -- sh -c "while true; do wget -O - -q http://kubia.default; done"
```

Она запустит модуль, который будет циклически делать запросы в службу `kubia`. Вы несколько раз встречали параметр `-it` при выполнении команды `kubectl exec`. Вы можете наглядно убедиться, что его также можно использовать с командой `kubectl run`. Она позволяет прикреплять к процессу консоль, которая не только будет вам показывать результат процесса напрямую, но и завершит процесс, как только вы нажмете Ctrl+C. Параметр `--rm` впоследствии приводит к удалению модуля, а параметр `--restart=Never` заставляет команду `kubectl run` создавать неуправляемый модуль напрямую, а не посредством объекта Deployment, который вам не нужен. Эта комбинация параметров полезна для выполнения команд внутри кластера без необходимости опираться на существующий модуль. Она не только ведет себя так же, как если бы вы запускали команду локально, но и даже все очищает, когда эта команда завершается.

Просмотр увеличения масштаба развертывания автопреобразователем

В ходе работы модуля-генератора загрузки вы увидите, что он первоначально будет попадать в единственный модуль. Как и раньше, для обновления метрик требуется время, но, когда они будут обновлены, вы увидите, что автопреобразователь масштаба увеличит количество реплик. В моем случае задействованность ЦП модуля первоначально подпрыгнула до 108%, что заставило автопреобразователь масштаба увеличить количество модулей до четырех. Задействованность на отдельных модулях затем снизилась до 74%, а потом стабилизировалась на уровне около 26%.

ПРИМЕЧАНИЕ. Если загрузка ЦП в вашем случае не превышает 30%, попробуйте запустить дополнительные генераторы загрузки.

Опять-таки, вы можете проинспектировать события автопреобразователя масштаба с помощью команды `kubectl describe` и увидеть результаты работы автопреобразователя (в следующем ниже листинге показана только самая важная информация).

Листинг 15.5. События автопреобразователя масштаба HorizontalPodAutoscaler

```
From  Reason             Message
----  ------             -------
h-p-a  SuccessfulRescale  New size: 1; reason: All metrics below target
h-p-a  SuccessfulRescale  New size: 4; reason: cpu resource utilization
                          (percentage of request) above target
```

Не кажется ли вам странным, что первоначальная средняя задействованность ЦП в моем случае, когда у меня был всего один модуль, составила 108%, превысив 100%? Напомним, что задействованность ЦП контейнера – это фактическое потребление ЦП контейнера, деленное на запрошенный ЦП. Запрошенный ЦП определяет минимальный, а не максимальный объем ЦП, доступный контейнеру, и поэтому контейнер может потреблять больше, чем запрошенный ЦП, в результате чего процент превышает 100.

Прежде чем мы продолжим, давайте сделаем небольшие расчеты и посмотрим на то, как автопреобразователь масштаба пришел к выводу, что требуется четыре реплики. Первоначально имелась одна реплика, которая занималась запросами, и ее потребление ЦП увеличилось до 108%. В результате деления 108 на 30 (на целевой процент задействованности процессора) получается 3.6, и это число затем округляется автопреобразователем до 4. Если вы разделите 108 на 4, то получите 27%. Если автопреобразователь увеличивает масштаб до четырех модулей, то их средняя задействованность ЦП ожидается где-то в окрестности 27%, что близко к целевому значению в 30% и почти в точности соответствует наблюдавшейся задействованности ЦП.

Максимальная скорость масштабирования

В моем случае потребление ЦП выстрельнуло до 108%, но в общем случае первоначальное потребление ЦП может возрасти еще выше. Даже если первоначальная средняя задействованность ЦП была выше (скажем, 150%), требуя для достижения цели в 30% пяти реплик, автопреобразователь масштаба на первом шаге все равно будет увеличивать масштаб только до четырех модулей, потому что он имеет лимит на то, какое количество реплик можно добавлять в одной операции масштабирования. Если существует более двух текущих реплик, то автопреобразователь за одну операцию увеличения масштаба будет не более чем удваивать количество реплик. Если существует только одна или две реплики, он будет увеличивать масштаб до максимума из четырех реплик за один шаг.

Кроме того, он имеет лимит на то, как скоро может произойти последующая операция автомасштабирования после предыдущей. В настоящее время уве-

личение масштаба выполняется только в том случае, если событие перемасштабирования не произошло за последние три минуты. Событие уменьшения масштаба наступает еще реже – каждые пять минут. Имейте это в виду и не удивляйтесь, почему автопреобразователь масштаба отказывается выполнять операцию перемасштабирования, даже если на это явно указывают метрики.

Изменение целевого метрического значения для существующего объекта HPA

В завершение этого раздела давайте выполним одно последнее упражнение. Возможно, ваша первоначальная цель задействованности ЦП в 30% была слишком низкой, поэтому увеличьте ее до 60%. Для этого отредактируйте ресурс HPA с помощью команды `kubectl edit`. Когда откроется текстовый редактор, измените значение поля `targetAverageUtilization` на 60, как показано в следующем ниже листинге.

Листинг 15.6. Увеличение целевой задействованности ЦП путем правки ресурса HPA

```
...
spec:
  maxReplicas: 5
  metrics:
  - resource:
      name: cpu
      targetAverageUtilization: 60
    type: Resource
...
```

Поменяйте это с 30 на 60

Как и в случае с большинством других ресурсов, после модификации ресурса ваши изменения будут обнаружены контроллером автопреобразователя масштаба и обработаны. Кроме того, вы можете удалить ресурс и повторно создать его с другими целевыми значениями, поскольку, удаляя ресурс HPA, вы лишь деактивируете автоматическое масштабирование целевого ресурса (в данном случае ресурса развертывания Deployment) и оставляете его в том масштабе, в котором он находился в это время. Автоматическое масштабирование возобновится после создания нового ресурса HPA для развертывания Deployment.

15.1.3 Масштабирование на основе потребления памяти

Вы увидели, как легко конфигурируется автопреобразователь горизонтального масштаба, для того чтобы удерживать задействованность ЦП на целевом уровне. Но как насчет автомасштабирования на основе потребления модулями памяти?

Выполнять автомасштабирование на основе памяти гораздо проблематичнее, чем автомасштабирование на основе ЦП. Основная причина заключается в том, что после масштабирования старые модули каким-то образом должны

принудительно освободить память. Это должно быть сделано самим приложением, а не системой. Система может лишь уничтожить и перезапустить приложение, надеясь, что оно будет использовать меньше памяти, чем раньше. Но если приложение затем использует тот же объем, что и раньше, то автопреобразователь масштаба увеличит его масштаб снова. И затем еще раз – и до тех пор, пока он не достигнет максимального количества модулей, сконфигурированных на ресурсе HPA. Совершенно очевидно, это совсем не то, что нужно. Автомасштабирование на основе памяти было введено в Kubernetes версии 1.8 и настроено точно так же, как автомасштабирование на основе ЦП. Исследование его работы оставлено на усмотрение читателя.

15.1.4 Масштабирование на основе других, а также настраиваемых метрик

Вы видели, как легко масштабировать модули на основе их потребления ЦП. Первоначально это был единственный вариант автомасштабирования, который можно было использовать на практике. Сделать так, что автопреобразователь масштаба использовал настраиваемые, определяемые приложением метрики, которые бы направляли его масштабирующие решения, было довольно сложно. Первоначальная конструкция автопреобразователя масштаба не позволяет легко перейти от простого масштабирования на основе ЦП. Это побудило специальную группу по автомасштабированию Kubernetes Autoscaling Special Interest Group (SIG) полностью перепроектировать автопреобразователь масштаба.

Если вы заинтересованы узнать, насколько сложным было применение исходного автопреобразователя масштаба с настраиваемыми метриками, я приглашаю вас прочитать мой пост в блоге под названием «Автоматическое масштабирование в Kubernetes на основе специальных показателей без использования порта хоста», который вы найдете в интернете по http://medium.com/@marko.luksa. Вы также познакомитесь со всеми другими проблемами, с которыми я столкнулся при попытке настроить автоматическое масштабирование на основе специальных показателей. К счастью, новые версии Kubernetes этих проблем не имеют. И я расскажу об этом в новом блог-посте.

Вместо того чтобы привести здесь законченный пример, давайте кратко рассмотрим, как сконфигурировать автопреобразователь масштаба для использования различных источников метрик. Мы начнем с исследования того, как мы определили, какую метрику использовать в нашем предыдущем примере. В следующем ниже листинге показано, как предыдущий объект HPA был сконфигурирован для использования показателя потребления ЦП.

Листинг 15.7. Определение автопреобразователя HorizontalPodAutoscaler для автомасштабирования на основе ЦП

```
...
spec:
```

```
  maxReplicas: 5
  metrics:
  - type: Resource              <-- Определяет тип метрики
    resource:
      name: cpu                 <-- Ресурс, задействование которого будет отслеживаться
      targetAverageUtilization: 30   <-- Целевая задействованность этого ресурса
...
```

Как видно, поле `metrics` позволяет определять несколько используемых метрик. В данном листинге используется одна метрика. Каждая запись определяет тип метрики – в данном случае метрика `Resource`. В объекте HPA можно использовать три типа метрик:

- `Resource`;
- `Pods`;
- `Object`.

Тип метрики Resource

Тип `Resource` заставляет автопреобразователь масштаба основывать свои масштабирующие решения на ресурсной метрике, наподобие тех, которые указаны в ресурсных запросах контейнера. Мы уже видели, как это делается, поэтому давайте сосредоточимся на двух других типах.

Тип метрики Pods

Тип `Pods` используется для ссылки на любую другую (включая настраиваемую) метрику, относящуюся непосредственно к модулю. Примером такой метрики могут быть уже упомянутые запросы в секунду (QPS) или количество сообщений в очереди брокера сообщений (когда брокер сообщений работает как модуль). Для того чтобы сконфигурировать автопреобразователь масштаба для использования метрики QPS модуля, объект HPA должен включать запись в поле `metrics`, показанную в следующем ниже листинге.

Листинг 15.8. Ссылка на настраиваемую метрику модуля в HPA

```
...
spec:
  metrics:
  - type: Pods                  <-- Определяет метрику модуля
    resource:
      metricName: qps           <-- Имя метрики
      targetAverageValue: 100   <-- Среднее целевое значение для всех целевых модулей
...
```

Пример в этом листинге настраивает автопреобразователь масштаба на удержание среднего значения QPS всех модулей, управляемых контроллером

ReplicaSet (или другим контроллером), являющихся целью для этого ресурса HPA, на уровне `100`.

Тип метрики Object

Тип метрики `Object` используется, когда вам нужно заставить автопреобразователь масштаба масштабировать модули на основе метрики, которая не относится непосредственно к этим модулям. Например, может потребоваться промасштабировать модули в соответствии с метрикой другого объекта кластера, в частности объекта Ingress. Метрикой может быть QPS, как показано в листинге 15.8, средняя задержка запроса или что-то еще.

В отличие от предыдущего случая, где автопреобразователю масштаба необходимо было получать метрики для всех целевых модулей, после чего брать среднее из этих значений, при использовании типа метрики `Object` автопреобразователь масштаба получает единственную метрику от единственного объекта. В определении HPA необходимо указать целевой объект и целевое значение. Соответствующий пример приведен в следующем ниже листинге.

Листинг 15.9. Ссылка на метрику другого объекта в HPA

```
...
spec:
  metrics:
  - type: Object
    resource:
      metricName: latencyMillis
      target:
        apiVersion: extensions/v1beta1
        kind: Ingress
        name: frontend
      targetValue: 20
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: kubia
...
```

Использовать метрику конкретного объекта (- type: Object)

Имя метрики (metricName: latencyMillis)

Конкретный объект, метрику которого должен получить автопреобразователь масштаба (target)

Автопреобразователь масштаба должен масштабировать таким образом, чтобы значение показателя оставалось близким к этому (targetValue: 20)

Масштабируемый ресурс, который автопреобразователь будет масштабировать (scaleTargetRef)

В данном примере объект HPA сконфигурирован на использование метрики `latencyMillis` объекта Ingress `frontend`. Целевое значение метрики равно `20`. Автопреобразователь горизонтального масштаба модулей будет отслеживать метрику объекта Ingress, и если она поднимается слишком высоко над целевым значением, то автопреобразователь промасштабирует ресурс Deployment `kubia`.

15.1.5 Определение метрик, подходящих для автомасштабирования

Необходимо понимать, что не все метрики подходят для использования в качестве основы автомасштабирования. Как отмечалось ранее, потребление памяти контейнерами модулей не является хорошим показателем для автомасштабирования. Автопреобразователь масштаба не будет функционировать должным образом, если увеличение количества реплик не приведет к линейному уменьшению среднего значения наблюдаемой метрики (или, по крайней мере, близкому к линейному).

Например, если у вас есть только один экземпляр модуля, и значение показателя равняется X, и автопреобразователь увеличивает масштаб до двух реплик, то эта метрика должна упасть до значения где-то приблизительно X/2. Примером такой настраиваемой метрики являются запросы в секунду (Queries per Second, QPS), которая в случае веб-приложений сообщает количество запросов, получаемых приложением в секунду. Увеличение количества реплик всегда приведет к пропорциональному уменьшению QPS, поскольку большее количество модулей будет обрабатывать одинаковое общее количество запросов.

Прежде чем вы решите основывать автопреобразователь масштаба на собственных метриках вашего приложения, обязательно подумайте о том, как его значение будет себя вести, когда количество модулей увеличивается или уменьшается.

15.1.6 Уменьшение масштаба до нуля реплик

Автопреобразователь горизонтального масштаба модулей в настоящее время не позволяет устанавливать поле `minReplicas` равным 0, поэтому автопреобразователь никогда не будет уменьшать масштаб до нуля, даже если модули ничего не делают. Возможность уменьшения количества модулей до нуля может значительно увеличить задействованность вашего оборудования. Когда вы запускаете службы, которые получают запросы всего один раз каждые несколько часов или даже дней, нет смысла в том, чтобы они работали все время, съедая ресурсы, которые могут использоваться другими модулями. Но вы все равно хотите, чтобы эти службы были доступны сразу же, когда поступает запрос от клиента.

Это называется режимом холостого хода и выходом из него. Он позволяет уменьшать до нуля масштаб модулей, которые обеспечивают некую службу. Когда поступает новый запрос, этот запрос блокируется до тех пор, пока модуль не будет поднят, и затем запрос окончательно перенаправляется в модуль.

Kubernetes в настоящее время пока этого функционала не предоставляет, но он будет в конечном итоге предоставлен. Проверьте документацию относительно реализации режима холостого хода (idling).

15.2 Вертикальное автомасштабирование модуля

Возможность выполнять горизонтальное масштабирование сама по себе замечательная, но не каждое приложение можно промасштабировать по горизонтали. Для таких приложений единственный вариант – масштабировать их по вертикали – предоставить им больше ЦП и/или памяти. Поскольку узел обычно имеет больше ресурсов, чем запросы одного модуля, почти всегда можно промасштабировать модуль по вертикали, верно?

Поскольку ресурсные запросы модуля конфигурируются с помощью полей в манифесте модуля, вертикальное масштабирование модулей будет выполняться путем изменения этих полей. Я говорю «будет», потому что в настоящее время отсутствует возможность изменить ресурсные запросы и лимиты существующих модулей. До того, как я начал писать книгу (более года назад), я был уверен, что к тому времени, когда я напишу эту главу, Kubernetes уже будет поддерживать правильное вертикальное масштабирование модулей, поэтому я включил его в предварительное оглавление книги. К сожалению, вертикальное масштабирование модуля пока еще недоступно, и его реализация все время откладывается на более поздний срок.

15.2.1 Автоматическое конфигурирование ресурсных запросов

Экспериментальный функционал устанавливает запросы на ЦП и память на вновь создаваемых модулях, если их контейнеры не имеют их в явном виде. Этот функционал предоставляется плагином управления допуском InitialResources. При создании нового модуля без ресурсных запросов модуль просматривает исторические данные о потреблении ресурсов контейнерами модуля (в соответствии с базовым образом контейнера и его тегом) и устанавливает соответствующие запросы.

Вы можете развертывать модули без указания ресурсных запросов и опираться на Kubernetes, который в конечном итоге определит потребности в ресурсах каждого контейнера. Фактически Kubernetes вертикально масштабирует модуль. Например, если в контейнере не хватает памяти, то при следующем создании модуля с этим образом контейнера его ресурсный запрос на память будет автоматически установлен выше.

15.2.2 Модификация ресурсных запросов во время работы модуля

В конечном итоге тот же механизм будет использоваться для изменения ресурсных запросов существующего модуля, что означает вертикальное масштабирование модуля во время его работы. Во время написания этой книги новое предложение по вертикальному масштабированию модуля находилось в стадии завершения. Относительно реализации автоматического преобразо-

вания вертикального масштаба модулей, пожалуйста, обратитесь к документации Kubernetes.

15.3 Горизонтальное масштабирование узлов кластера

Автоматический преобразователь вертикального масштаба модулей создает дополнительные экземпляры модуля, когда в них возникает потребность. Но как быть, когда все узлы находятся на уровне максимальной загрузки и не способны запускать дополнительные модули? Совершенно очевидно, что эта проблема не ограничивается только тем, когда автопреобразователь масштаба создает новые экземпляры модуля. Даже при создании модулей вручную может возникнуть проблема, когда ни один из узлов не может принять новые модули, поскольку ресурсы узла израсходованы существующими модулями.

В этом случае необходимо удалить несколько существующих модулей, промасштабировать их по вертикали или добавить дополнительные узлы в кластер. Если кластер Kubernetes работает локально, то вам нужно физически добавить новую машину и сделать ее частью кластера Kubernetes. Но если кластер работает в облачной инфраструктуре, то добавление дополнительных узлов обычно происходит в несколько щелчков мыши или вызовом API в облачной инфраструктуре. Это делается автоматически, верно?

Система Kubernetes включает в себя функционал автоматического запроса дополнительных узлов у поставщика облачных служб, как только она обнаруживает необходимость в дополнительных узлах. Это выполняется кластерным автопреобразователем масштаба.

15.3.1 Знакомство с кластерным автопреобразователем масштаба

Кластерный автопреобразователь масштаба занимается автоматическим резервированием дополнительных узлов, когда он замечает модуль, который не может быть назначен на существующие узлы из-за нехватки ресурсов на этих узлах. Он также отключает узлы, когда они используются недостаточно в течение длительных периодов времени.

Запрос дополнительных узлов у облачной инфраструктуры

Новый узел будет зарезервирован, если после создания нового модуля планировщик не сможет назначить его ни на один из существующих узлов. Кластерный автопреобразователь масштаба высматривает такие модули и запрашивает у поставщика облачных служб запустить дополнительный узел. Но прежде чем сделать это, он проверяет, сможет ли новый узел вообще разместить модуль. В конце концов, если это не так, нет смысла запускать такой узел.

Поставщики облачных служб обычно объединяют узлы в группы (или пулы) узлов одного размера (или узлов с одинаковым функционалом). Следовательно, кластерный автопреобразователь масштаба не может просто заявить «Дайте мне дополнительный узел». Необходимо также указать тип узла.

Кластерный автопреобразователь масштаба делает это, исследуя доступные группы узлов, чтобы увидеть, сможет ли, по крайней мере, один из типов узла соответствовать неназначенному модулю. Если существует ровно одна такая группа узлов, то кластерный автопреобразователь масштаба может увеличить размер группы узлов, чтобы поставщик облачных служб добавил еще один узел в группу. Если имеется несколько вариантов, то кластерный автопреобразователь масштаба должен выбрать самый лучший. Точное значение слова «лучший», очевидно, подразумевает конфигурируемость. В худшем случае он выбирает случайный вариант. Простой обзор того, как кластерный автопреобразователь масштаба реагирует на неназначенный модуль, показан на рис. 15.5.

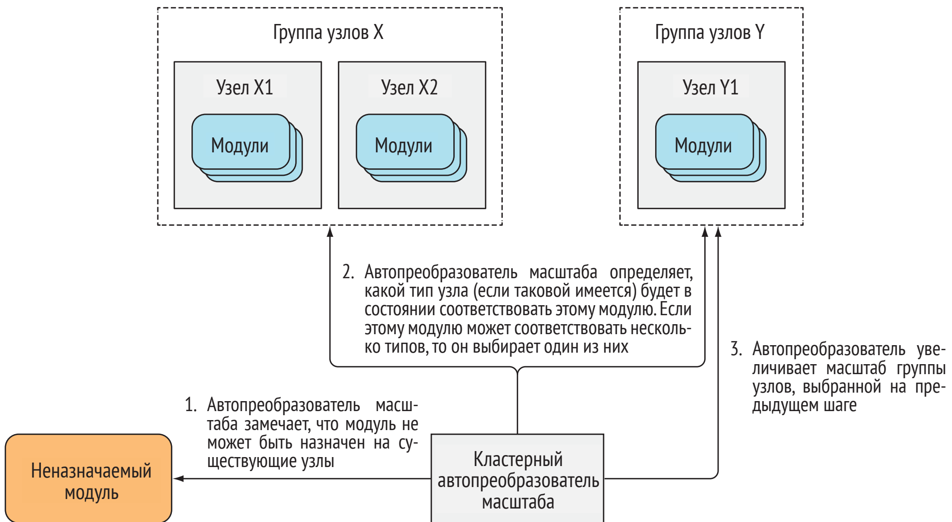


Рис. 15.5. Кластерный автопреобразователь увеличивает масштаб при обнаружении модуля, который не может быть назначен на существующие узлы

При запуске нового узла агент Kubelet на этом узле связывается с сервером API и регистрирует узел, создавая ресурс узла Node. С этого момента узел является частью кластера Kubernetes, и на него можно назначать модули.

Просто, правда? А что насчет уменьшения масштаба?

Отказ от узлов

Кластерный автопреобразователь масштаба также должен уменьшать количества узлов, когда они не задействуются в достаточной мере. Автопреобразователь масштаба делает это, отслеживая запрошенные ЦП и память на всех узлах. Если запросы на ЦП и память всех модулей, работающих на данном узле, ниже 50%, то узел считается ненужным.

Это не единственный определяющий фактор при принятии решения о том, чтобы отказаться от узла. Автопреобразователь масштаба также проверяет, работают ли какие-либо системные модули (только) на этом узле (кроме тех, которые выполняются на каждом узле, потому что они развертываются, например, набором демонов DaemonSet). Если на узле работает системный модуль, то этот узел не будет отменен. То же самое верно и в том случае, если на узле выполняется неуправляемый модуль или модуль с локальным хранилищем, так как это приведет к нарушению работы службы, предоставляемой модулем. Другими словами, узел будет возвращен поставщику облака только в том случае, если кластерный автопреобразователь масштаба знает, что модули, работающие на узле, будут переназначены на другие узлы.

Когда узел, который должен быть закрыт, выбран, этот узел сначала помечается как неназначаемый, и тогда все работающие на узле модули вытесняются. Поскольку все эти модули принадлежат набору реплик ReplicaSets или другим контроллерам, создаются сменные модули и назначаются остальным узлам (именно по этой причине узел, который будет закрыт, сначала помечается как неназначаемый).

Ручное блокирование и опустошение узлов

Узел также может быть помечен как неназначаемый и опустошен вручную. Не вдаваясь в подробности, это делается с помощью следующих ниже команд `kubectl`:

- `kubectl cordon <узел>` помечает узел как неназначаемый (но не делает ничего с модулями, работающими на этом узле);
- `kubectl drain <узел>` помечает узел как неназначаемый, а затем вытесняет все модули узла.

В обоих случаях никакие новые модули узлу не назначаются до тех пор, пока вы снова не разблокируете его командой `kubectl uncordon <узел>`.

15.3.2 Активация кластерного автопреобразователя масштаба

Кластерный автопреобразователь масштаба в настоящее время имеется на:

- Google Kubernetes Engine (GKE);
- Google Compute Engine (GCE);
- Amazon Web Services (AWS);
- Microsoft Azure.

То, как вы запускаете автопреобразователь масштаба, зависит от того, где находится ваш кластер Kubernetes. Для кластера `kubia`, работающего на GKE, кластерный автопреобразователь масштаба можно активировать следующим образом:

```
$ gcloud container clusters update kubia --enable-autoscaling \
  --min-nodes=3 --max-nodes=5
```

Если кластер работает на GCE, то перед запуском `kube-up.sh` необходимо задать три переменные среды:

- `KUBE_ENABLE_CLUSTER_AUTOSCALER=true`;
- `KUBE_AUTOSCALER_MIN_NODES=3`;
- `KUBE_AUTOSCALER_MAX_NODES=5`.

Обратитесь к репозиторию кластерного автопреобразователя масштаба на Github по адресу https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler для получения информации о том, как его активировать на других платформах.

> **ПРИМЕЧАНИЕ.** Кластерный автопреобразователь масштаба публикует свой статус в словарь конфигурации ConfigMap `cluster-autoscaler-status` в пространстве имен `kube-system`.

15.3.3 Ограничение прерывания службы во время уменьшения масштаба кластера

Когда узел прекращает свою работу неожиданно, ничто не может помешать его модулям стать недоступными. Но когда узел завершает работу добровольно, либо кластерным автопреобразователем масштаба, либо человеком-оператором, вы можете обеспечить, чтобы эта операция не прерывала работу службы, предоставляемой модулями, работающими на этом узле. Это делается посредством дополнительного функционала.

Некоторые службы требуют, чтобы всегда работало минимальное количество модулей; это особенно верно для кластерных приложений на основе кворума. По этой причине Kubernetes предоставляет способ определения минимального количества модулей, которые должны продолжать работать при выполнении этих типов операций. Это делается путем создания ресурса PodDisruptionBudget.

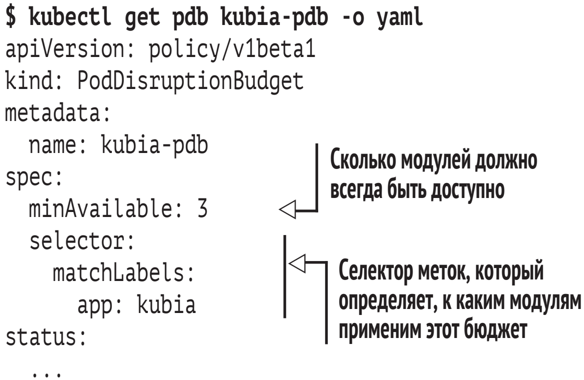
Несмотря на то что название ресурса звучит сложно (бюджет прерывания работы модулей), это один из самых простых из доступных ресурсов Kubernetes. Он содержит только селектор меток модуля и число, указывающее на минимальное количество модулей, которые всегда должны быть доступны, или, начиная с версии kubernetes 1.7, максимальное количество модулей, которые могут быть недоступны. Мы рассмотрим, как выглядит манифест ресурса PodDisruptionBudget (PDB), но, вместо того чтобы создавать его из файла YAML, вы создадите его с помощью команды `kubectl create poddisruptionbudget`, а затем получите и исследуете его YAML позже.

Если вы хотите гарантировать, чтобы всегда работали три экземпляра модуля `kubia` (у них есть метка `app=kubia`), создайте ресурс PodDisruptionBudget следующим образом:

```
$ kubectl create pdb kubia-pdb --selector=app=kubia --min-available=3
poddisruptionbudget "kubia-pdb" created
```

Просто, правда? Теперь получите файл YAML ресурса PDB. Это показано в следующем ниже листинге.

Листинг 15.10. Определение PodDisruptionBudget



В поле `minAvailable` можно также использовать процент вместо абсолютного числа. Например, можно указать, что 60% всех модулей с меткой `app=kubia` должны работать в любое время.

> **ПРИМЕЧАНИЕ.** Начиная с версии Kubernetes 1.7 ресурс PodDisruptionBudget также поддерживает поле `maxUnavailable`, которое можно использовать вместо `minAvailable`, в случае если требуется блокировать вытеснения из узла, когда недоступно больше, чем указанное количество модулей.

Об этом ресурсе особо нечего сказать. Пока он существует, кластерный автопреобразователь масштаба и команда `kubectl drain` будут придерживаться его и никогда не будут вытеснять модуль с меткой `app=kubia`, если это приведет к тому, что количество таких модулей станет ниже трех.

Например, если всего было четыре модуля и полю `minAvailable` было присвоено значение три, как в данном примере, то процесс вытеснения модулей будет вытеснять модули один за другим, ожидая, когда контроллер ReplicaSet заменит вытесненный модуль новым, и только потом приступит к вытеснению следующего модуля.

15.4 Резюме

В этой главе было показано, как Kubernetes может масштабировать не только ваши модули, но и ваши узлы. Вы узнали, что:

- процедура конфигурирования автоматического преобразователя горизонтального масштаба проста и сводится к созданию объекта Horizon-

talPodAutoscaler и указанию его на развертывание Deployment, набор реплик ReplicaSet или контроллер репликации ReplicationController и назначению модулям показателя целевой задействованности ЦП;
- помимо того что автопреобразователь горизонтального масштаба выполняет масштабирующие операции на основе задействованности ЦП модулями, вы также можете настроить его для масштабирования на основе своих собственных прикладных настраиваемых метрик или метрик, связанных с другими объектами, развернутыми в кластере;
- вертикальное масштабирование модуля пока невозможно;
- если кластер Kubernetes работает на поддерживаемом облачном провайдере, то автоматически масштабировать можно даже узлы кластера;
- в модуле можно выполнять разовые процессы и автоматически останавливать и удалять модуль нажатием Ctrl+C, используя команду `kubectl run` с параметрами `-it` и `--rm`.

В следующей далее главе мы рассмотрим расширенный функционал назначения модулей узлам, в частности как держать определенные модули подальше от определенных узлов и как назначать модули близко друг к другу или далеко друг от друга.

Глава 16

Продвинутое назначение модулей узлам

Эта глава посвящена:

- использованию ограничений узлов и допусков модулей, для того чтобы держать модули на удалении от некоторых узлов;
- определению правил сходства узлов в качестве альтернативы селекторам узлов;
- совместному размещению модулей с использованием сходства модулей;
- удержанию модулей на удалении друг от друга с использованием антисходства модулей.

Kubernetes позволяет влиять на то, куда модули назначаются. Первоначально это делалось только путем задания селектора узлов в спецификации модуля, но позже были добавлены дополнительные механизмы, которые расширили эту функциональность. Они рассматриваются в этой главе.

16.1 Использование ограничений и допусков для отделения модулей от определенных узлов

Первые два функциональных средства, связанных с расширенным назначением модулей узлам, которые мы здесь рассмотрим, – это ограничения узлов и допуски модулей. Они используются для ограничения того, какие модули могут использовать определенный узел. Модуль может быть назначен узлу, только если он допускает дефекты узла.

Это несколько отличается от использования селекторов узлов и сходства узлов, о которых вы узнаете далее в данной главе. Селекторы узлов и правила сходства узлов позволяют выбирать то, каким узлам модуль может или не может быть назначен, специально добавляя эту информацию в модуль, в

то время как ограничения позволяют отклонять развертывание модулей на определенных узлах, лишь добавляя в узел ограничения без необходимости изменять существующие модули. Модули, которые вы хотите видеть развернутыми на узле с заданными допущениями, должны согласиться использовать такой узел, тогда как в случае с селекторами узлов модули явным образом указывают, на каком узле (узлах) они хотят быть развернуты.

16.1.1 Знакомство с ограничениями и допусками

Лучший способ познакомиться с ограничениями узлов – увидеть существующее ограничение. В приложении B показано, как настроить многоузловой кластер с помощью инструмента `kubeadm`. По умолчанию ведущий узел в таком кластере ограничен, поэтому на нем могут быть развернуты только модули плоскости управления.

Вывод на экран ограничений узла

Дефекты узла можно увидеть с помощью команды `kubectl describe node`, как показано в следующем ниже листинге.

Листинге 16.1. Описание ведущего узла в кластере, созданном инструментом kubeadm

```
$ kubectl describe node master.k8s
Name:          master.k8s
Role:
Labels:        beta.kubernetes.io/arch=amd64
               beta.kubernetes.io/os=linux
               kubernetes.io/hostname=master.k8s
               node-role.kubernetes.io/master=
Annotations:   node.alpha.kubernetes.io/ttl=0
               volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:        node-role.kubernetes.io/master:NoSchedule
...
```


Ведущий узел имеет одно ограничение. Ограничения имеют *ключ*, *значение* и *проявление* и представляются в формате `<ключ>=<значение>:<проявление>`. Ограничение ведущего узла, показанное в приведенном выше листинге, имеет ключ `node-role.kubernetes.io/master`, значение `null` (в дефекте не показано) и проявление `NoSchedule`.

Это ограничение не дает модулям быть назначенными ведущему узлу, если только эти модули не допускают данного дефекта. Модули, которые его допускают, обычно являются системными модулями (см. рис. 16.1).

Вывод на экран допущений

В кластере, установленном с помощью `kubeadm`, кластерный компонент kube-proxy выполняется как модуль на каждом узле, включая ведущий узел,

поскольку ведущие компоненты, работающие как модули, также могут нуждаться в доступе к службам Kubernetes. Для того чтобы убедиться, что модуль kube-proxy также запустится на ведущем узле, он включает в себя соответствующие допуски. В общей сложности модуль имеет три допуска, которые показаны в следующем ниже листинге.

Листинг 16.2. Допущения модуля

```
$ kubectl describe po kube-proxy-80wqm -n kube-system
...
Tolerations: node-role.kubernetes.io/master=:NoSchedule
node.alpha.kubernetes.io/notReady=:Exists:NoExecute
node.alpha.kubernetes.io/unreachable=:Exists:NoExecute
...
```

Как вы можете видеть, первый допуск совпадает с ограничением ведущего узла, позволяя модулю kube-proxy быть назначенным главному узлу.

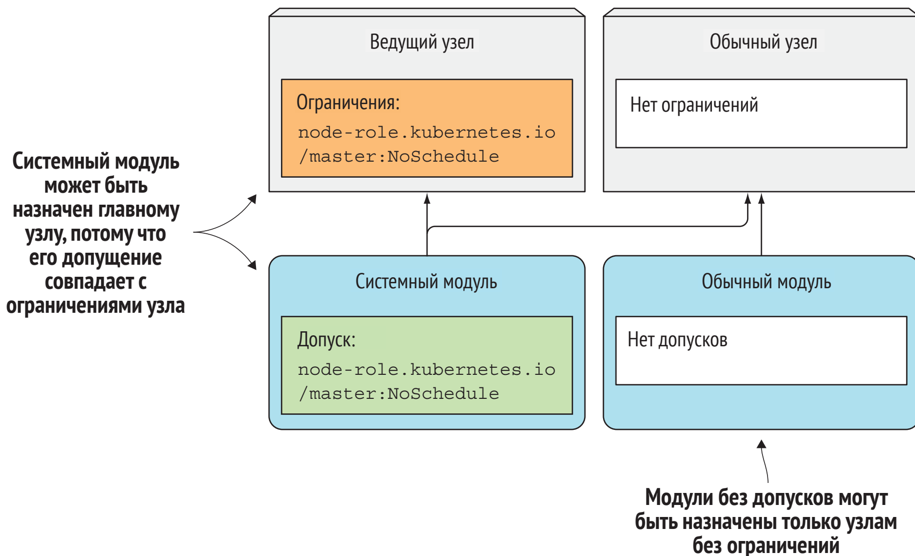


Рис. 16.1. Модуль назначается узлу, только если он допускает дефекты узла

ПРИМЕЧАНИЕ. Не обращайте внимания на знак равенства, который показан в допущениях модуля; он не относится к ограничениям узла. Когда значение ограничения/допуска равно `null`, агент Kubectl, по-видимому, отображает ограничения и допуски по-разному.

Проявления ограничений

Два других допуска в модуле kube-proxy определяют, как долго модуль может работать на неготовых или недоступных узлах (время в секундах не пока-

зано, но его можно увидеть в YAML модуля). Эти два допущения относятся к проявлению `NoExecute` вместо проявления `NoSchedule`.

С каждым ограничением связано его проявление. Существует три возможных проявления:

- `Noschedule`, которое значит, что модули не будут назначены узлу, если они не допускают ограничения;
- `PreferNoSchedule` – это мягкая версия `NoSchedule`, которая означает, что планировщик попытается избежать назначения модуля узлу, но назначит его узлу, если он не может назначить его где-то еще;
- `NoExecute`, в отличие от проявлений `NoSchedule` и `PreferNoSchedule`, которые влияют только на назначение модуля узлу, также влияет на модули, уже работающие на узле. Если в узел добавить ограничение `NoExecute`, то модули, которые уже работают на этом узле и не допускают ограничения `NoExecute`, будут вытеснены из узла.

16.1.2 Добавление в узел индивидуально настроенных ограничений

Представьте себе один кластер Kubernetes, в котором выполняются как задачи рабочего, так и нерабочих окружений. Крайне важно, чтобы модули никогда не запускались на узлах рабочего окружения. Это может быть достигнуто путем добавления ограничения в узлы рабочего окружения. Для добавления ограничения используется команда `kubectl taint`:

```
$ kubectl taint node node1.k8s node-type=production:NoSchedule
node "node1.k8s" tainted
```

Эта команда добавляет ограничение с ключом `node-type`, значением `production` и проявлением `NoSchedule`. Если теперь вы развернете несколько реплик обычного модуля, то увидите, что ни одна из них не назначена узлу, который вы пометили ограничением, как показано в следующем ниже листинге.

Листинг 16.3. Развертывание модулей без допуска

```
$ kubectl run test --image busybox --replicas 5 -- sleep 99999
deployment "test" created

$ kubectl get po -o wide
NAME                READY  STATUS   RESTARTS  AGE  IP         NODE
test-196686-46ngl   1/1    Running  0         12s  10.47.0.1  node2.k8s
test-196686-73p89   1/1    Running  0         12s  10.47.0.7  node2.k8s
test-196686-77280   1/1    Running  0         12s  10.47.0.6  node2.k8s
test-196686-h9m8f   1/1    Running  0         12s  10.47.0.5  node2.k8s
test-196686-p85ll   1/1    Running  0         12s  10.47.0.4  node2.k8s
```

Теперь никто не сможет случайно развернуть модули на узлах рабочего окружения.

16.1.3 Добавление в модули допусков

Для того чтобы развернуть производственные модули на узлы рабочего окружения, они должны допускать добавленные в узлы ограничения. Манифесты ваших производственных модулей должны включать фрагмент YAML, показанный в следующем ниже листинге.

Листинг 16.4. Производственное развертывание с допущением: production-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: prod
spec:
  replicas: 5
  template:
    spec:
      ...
      tolerations:
      - key: node-type
        Operator: Equal
        value: production
        effect: NoSchedule
```

Такой допуск позволяет назначать модуль узлам рабочего окружения

Если вы развернете этот ресурс Deployment, то увидите, что его модули будут развернуты на рабочем узле, как показано в следующем ниже листинге.

Листинг 16.5. Модули с допуском развертываются на узле рабочего окружения

```
$ kubectl get po -o wide
NAME                READY  STATUS   RESTARTS  AGE  IP          NODE
prod-350605-1ph5h   0/1    Running  0         16s  10.44.0.3   node1.k8s
prod-350605-ctqcr   1/1    Running  0         16s  10.47.0.4   node2.k8s
prod-350605-f7pcc   0/1    Running  0         17s  10.44.0.6   node1.k8s
prod-350605-k7c8g   1/1    Running  0         17s  10.47.0.9   node2.k8s
prod-350605-rp1nv   0/1    Running  0         17s  10.44.0.4   node1.k8s
```

Как видно из листинга, модули рабочего окружения также были развернуты на узле `node2`, который не является узлом рабочего окружения. Для того чтобы этого не происходило, вам также нужно пометить узлы разработки ограничением, в частности `node-type=Non-production:NoSchedule`. Затем вам также нужно добавить соответствующий допуск во все ваши модули разработки.

16.1.4 Для чего можно использовать ограничения и допуски

Узлы могут иметь несколько ограничений, а модули могут иметь несколько допусков. Как вы видели, ограничения могут иметь только ключ и проявление и не требуют значения. Допуски могут допускать определенное значение путем указания оператора `Equal` (который также задается по умолчанию, если он не указан), либо они могут допускать любое значение для определенного ключа ограничения, если используется оператор `Exists`.

Использование ограничений и допусков во время назначения модуля узлу

Ограничения могут использоваться для предотвращения назначения новых модулей (проявление `NoSchedule`) и определения непредпочтительных узлов (проявление `PreferNoSchedule`) и даже вытеснения существующих модулей из узла (`NoExecute`).

Вы можете настраивать ограничения и допуски любым удобным для вас способом. Например, можно разбить кластер на несколько секций, что позволит группам разработчиков планировать работу модулей только на соответствующих узлах. Вы также можете использовать ограничения и допуски, когда несколько ваших узлов предоставляют специальное оборудование и только часть ваших модулей должна его использовать.

Настройка времени, когда после аварийного отказа узла модуль будет переназначен

Допуски также можно использовать, чтобы указать, как долго Kubernetes должен ждать, прежде чем переназначить модуль на другой узел, если узел, на котором работает модуль, становится неготовым или недостижимым. Если вы посмотрите на допуски одного из ваших модулей, то увидите два допуска, которые показаны в следующем ниже листинге.

Листинг 16.6. Модуль с допусками, заданными по умолчанию

```
$ kubectl get po prod-350605-1ph5h -o yaml
...
  tolerations:
  - effect: NoExecute
    key: node.alpha.kubernetes.io/notReady
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.alpha.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
```

Модуль допускает узел, который не готов, в течение 300 секунд, после чего он будет переназначен

То же самое применяется к недостижимому узлу

Эти два допуска говорят о том, что данный модуль допускает неготовность узла или его недоступность в течение 300 секунд. Когда плоскость управления Kubernetes обнаруживает, что узел больше не готов или недоступен, она будет ждать 300 секунд, после чего удалит модуль и переназначит его на другой узел.

Эти два допуска автоматически добавляются в модули, которые их не определяют. Если для ваших модулей такая пятиминутная задержка слишком продолжительна, то вы можете ее сделать короче, добавив эти два допущения в секцию `spec` модуля.

> **ПРИМЕЧАНИЕ.** Данное функциональное средство в настоящее время находится в альфа-версии, и поэтому оно может измениться в будущих версиях Kubernetes. Вытеснения на основе ограничений также по умолчанию не активированы. Их можно активировать, запустив менеджер контроллеров с параметром `--feature-gates=TaintBasedEvictions=true`.

16.2 Использование сходства узлов для привлечения модулей к определенным узлам

Как вы узнали, ограничения используются, чтобы держать модули на удалении от определенных узлов. А сейчас вы узнаете о более новом механизме под названием «*сходство узлов*» (node affinity), который позволяет вам указывать Kubernetes назначать модули только определенным подмножествам узлов.

Сравнение сходства узлов с селекторами узлов

Первоначальный механизм сходства узлов в ранних версиях Kubernetes был полем `nodeSelector` в спецификации модуля. Для того чтобы иметь право стать целью для модуля, узел должен был включать все метки, указанные в этом поле.

Селекторы узлов хорошо справляются со своей работой и просты, но они не предлагают всего того, что вам может понадобиться. По этой причине был введен более мощный механизм. Селекторы узлов в конечном итоге будут объявлены устаревшими, поэтому крайне важно понимать новые правила сходства узлов.

Подобно селекторам узлов, каждый модуль может определять свои собственные правила сходства узлов. Они позволяют указывать жесткие требования или предпочтения. Указав предпочтение, вы сообщаете Kubernetes, какие узлы вы предпочитаете для определенного модуля, и Kubernetes попытается назначить модуль одному из таких узлов. Если это невозможно, он выберет один из других узлов.

Исследование стандартных меток узлов

Правило сходства узлов выбирает узлы, основываясь на их метках, так же, как это делают селекторы узлов. Прежде чем вы увидите, как использовать

сходство узлов, давайте рассмотрим метки одного из узлов в кластере Google Kubernetes Engine (GKE), чтобы разобраться в том, что такое стандартные метки узлов. Они показаны в следующем ниже листинге.

Листинг 16.7. Стандартные метки узла в GKE

```
$ kubectl describe node gke-kubia-default-pool-db274c5a-mjnf
Name:     gke-kubia-default-pool-db274c5a-mjnf
Role:
Labels:   beta.kubernetes.io/arch=amd64
          beta.kubernetes.io/fluentd-ds-ready=true
          beta.kubernetes.io/instance-type=f1-micro
          beta.kubernetes.io/os=linux
          cloud.google.com/gke-nodepool=default-pool
          failure-domain.beta.kubernetes.io/region=europe-west1
          failure-domain.beta.kubernetes.io/zone=europe-west1-d
          kubernetes.io/hostname=gke-kubia-default-pool-db274c5a-mjnf
```


Узел имеет много меток, но когда дело касается сходства узлов и сходства модулей, о котором вы узнаете позже, последние три являются наиболее важными. Значение этих трех меток выглядит следующим образом:

- `failure-domain.beta.kubernetes.io/region` определяет географический регион узла, в котором он находится;
- `failure-domain.beta.kubernetes.io/zone` определяет зону доступности, в которой узел находится;
- `kubernetes.io/hostname`, вполне понятно, является хостнеймом.

Эти и другие метки можно использовать в правилах сходства модулей. В главе 3 вы уже познакомились с тем, как добавлять свою собственную метку в узлы и использовать ее в селекторе узлов модуля. Своя собственная метка применяется для развертывания модулей только на узлах с этой меткой путем добавления селектора узлов в модули. Теперь вы увидите, как делать то же самое, используя правила сходства узлов.

16.2.1 Указание жестких правил сходства узлов

В примере, приведенном в главе 3, селектор узлов использовался для развертывания модуля, требующего GPU, только для узлов с GPU. Спецификация модуля включала поле `nodeSelector`, показанное в следующем ниже листинге.

Листинг 16.8. Модуль с использованием селектора узла: kubia-gpu-nodeselector.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
```

```
  nodeSelector:
    gpu: "true"
  ...
```

Этот модуль назначается только тем узлам, которые имеют метку gpu=true

Поле `nodeSelector` указывает на то, что этот модуль должен развертываться только на тех узлах, которые содержат метку `gpu=true`. Если заменить селектор узлов правилом сходства узлов, то определение модуля будет выглядеть следующим образом.

Листинг 16.9. Модуль с использованием правила nodeAffinity: kubia-gpu-nodeaffinity.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: gpu
            operator: In
            values:
            - "true"
```

Первое, что вы заметите, – это то, что оно гораздо сложнее, чем простой селектор узлов. Но это потому, что оно намного выразительнее. Рассмотрим данное правило подробнее.

Смысл длинного имени атрибута nodeAffinity

Как вы можете видеть, секция `spec` модуля содержит поле `affinity`, которое, в свою очередь, содержит поле `nodeAffinity`, а в нем содержится поле с очень длинным именем, поэтому давайте остановимся на первом.

Давайте разберем его на две части и рассмотрим, что они означают:

- `requiredDuringScheduling...` означает, что правила, определенные в этом поле, задают метки, которые узел должен иметь, чтобы модуль был назначен этому узлу;
- `...IgnoredDuringExecution` означает, что правила, определенные под этим полем, не влияют на модули, уже работающие на данном узле.

В этом месте давайте я вам немного упрощу вашу работу, сообщив, что сходство в настоящее время влияет только на назначение модуля узлу и никогда не приведет к вытеснению модуля из узла. Поэтому все правила всегда заканчиваются на `IgnoredDuringExecution`. В конечном счете Kubernetes также

будет поддерживать RequiredDuringExecution, то есть если вы удалите метку из узла, модули, которые требуют, чтобы узел имел эту метку, будут вытеснены из такого узла. Как я уже сказал, в Kubernetes это пока еще не поддерживается, поэтому давайте больше не будем беспокоиться о второй части этого длинного поля.

Поле nodeSelectorTerms

Держа в уме то, что было объяснено в предыдущем разделе, легко понять, что поле nodeSelectorTerms и поле matchExpressions определяют то, с какими выражениями должны совпадать метки узла, для того чтобы модуль был назначен узлу. Единственное выражение в данном примере понимается просто. Узел должен иметь метку gpu, значение которой равняется true.

Следовательно, этот модуль будет назначен только тем узлам, которые имеют метку gpu=true, как показано на рис. 16.2.

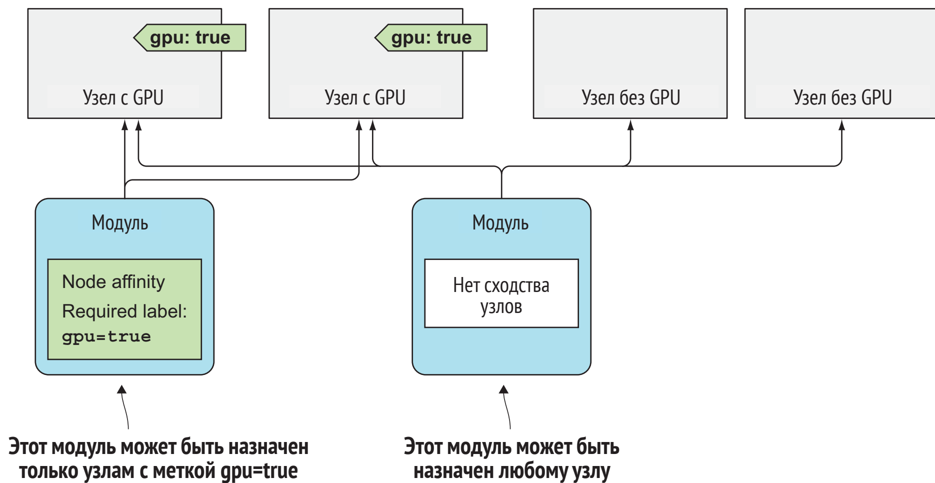


Рис. 16.2. Сходство узлов для модуля определяет, какие метки узел должен иметь, для того, чтобы модуль был ему назначен

Теперь наступает более интересная часть. Сходство узлов тоже позволяет устанавливать приоритеты узлов во время назначения модулей узлам. Мы рассмотрим это далее.

16.2.2 Приоретизация узлов при назначении модуля

Самое большое преимущество недавно введенного функционала сходства узлов – это возможность указывать на то, какие узлы планировщик должен предпочитать при назначении конкретного модуля. Это делается посредством поля preferredDuringSchedulingIgnoredDuringExecution.

Представьте себе наличие нескольких дата-центров в разных странах. Каждый дата-центр представляет отдельную зону доступности. В каждой зоне у вас есть определенные машины, предназначенные только для вашего собст-

венного использования, и другие, которые могут использоваться вашими компаниями-партнерами. Теперь вы хотите развернуть несколько модулей, и вы бы предпочли, чтобы они были назначены зоне `zone1` и машинам, зарезервированным для развертываний вашей компании. Если эти машины не имеют для этих модулей достаточно места или если существуют другие важные причины, которые мешают им быть там назначенными, то вы не возражаете, если они будут назначены машинам, которые используются вашими партнерами, и другим зонам. Сходство узлов позволяет это сделать.

Разметка узлов

Прежде всего узлы должны быть помечены соответствующим образом. Каждый узел должен иметь метку, обозначающую зону доступности, к которой принадлежит узел, и метку, помечающую его как выделенный или общий узел.

В приложении B объясняется, как настроить трехузловой кластер (один ведущий и два рабочих узла) в виртуальных машинах, работающих локально. В следующих далее примерах я буду использовать два рабочих узла в этом кластере, но вы также можете использовать Google Kubernetes Engine или любой другой многоузловой кластер.

> **ПРИМЕЧАНИЕ.** Minikube является не самым лучшим вариантом для этих примеров, потому что он выполняет только один узел.

Сначала пометьте узлы, как показано в следующем ниже листинге.

Листинг 16.10. Разметка узлов

```
$ kubectl label node node1.k8s availability-zone=zone1
node "node1.k8s" labeled
$ kubectl label node node1.k8s share-type=dedicated
node "node1.k8s" labeled
$ kubectl label node node2.k8s availability-zone=zone2
node "node2.k8s" labeled
$ kubectl label node node2.k8s share-type=shared
node "node2.k8s" labeled
$ kubectl get node -L availability-zone -L share-type
NAME         STATUS   AGE   VERSION   AVAILABILITY-ZONE   SHARE-TYPE
master.k8s   Ready    4d    v1.6.4    <none>              <none>
node1.k8s    Ready    4d    v1.6.4    zone1               dedicated
node2.k8s    Ready    4d    v1.6.4    zone2               shared
```

Указание правил предпочтительного сходства узлов

Настроив метки узлов, можно создать развертывание, которое предпочитает выделенные узлы `dedicated` в `zone1`. В следующем ниже листинге показан манифест развертывания.

Листинг 16.11. Развертывание с предпочтительным сходством узлов: preferred-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pref
spec:

  template:
    ...
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 80
            preference:
              matchExpressions:
              - key: availability-zone
                operator: In
                values:
                - zone1
          - weight: 20
            preference:
              matchExpressions:
              - key: share-type
                operator: In
                values:
                - dedicated
    ...
```

Вы указываете предпочтения, а не жесткие требования

Вы предпочитаете, чтобы модуль назначался зоне zone1. Это предпочтения являются самыми важными для вас

Вы также предпочитаете, чтобы ваши модули назначались выделенным узлам, но эти предпочтения в четыре раза менее важнее, чем предпочтение зоны

Давайте внимательно изучим этот листинг. Вы определяете предпочтение сходства узлов, а не жесткое требование. Вы хотите, чтобы модули назначались узлам, которые содержат метки `availability-zone=zone1` и `Share-type=dedicated`. Вы говорите, что первое правило предпочтения важно, установив его вес равным `80`, в то время как второе является гораздо менее важным (его вес равен `20`).

Как работают предпочтения узлов

Если в вашем кластере много узлов, то при назначении модулей развертывания в предыдущем листинге узлы будут разделены на четыре группы, как показано на рис. 16.3. Узлы, чьи метки `availability-zone` и `share-type` соответствуют сходству узлов модуля, занимают самую высокую позицию. Затем, вследствие того, как сконфигурированы веса в правилах сходства узлов модуля, далее идут узлы `shared` в зоне `zone1`, затем идут выделенные узлы `dedicated` в других зонах, и самый низкий приоритет имеют все остальные узлы.

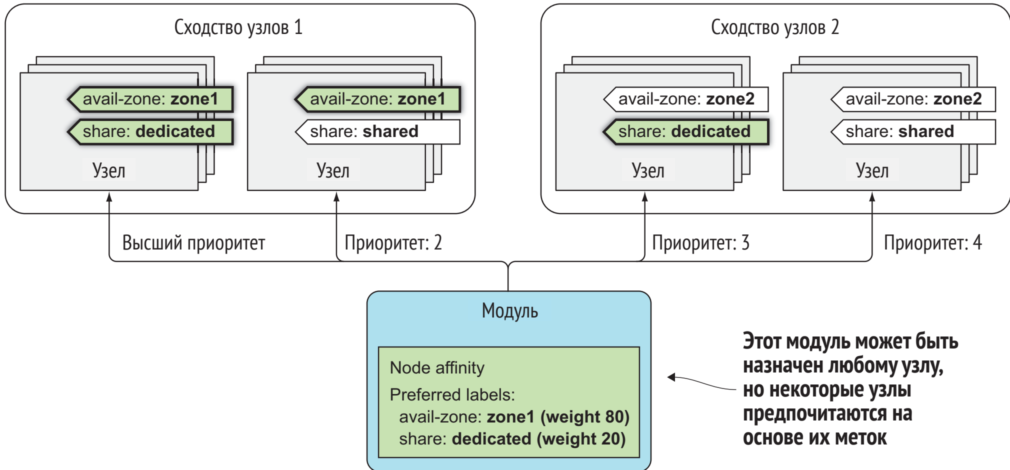


Рис. 16.3. Приоретизация узлов на основе предпочтений сходства узлов модуля

Развертывание модулей в двухузловом кластере

Если вы создадите это развертывание в своем двухузловом кластере, то вы увидите, что большинство модулей (если не все из них) развернуто в узле `node1`. Проинспектируйте следующий ниже листинг, чтобы выяснить, что это верно.

Листинг 16.12. Просмотр того, куда модули были назначены

```
$ kubectl get po -o wide
NAME                READY  STATUS   RESTARTS  AGE  IP         NODE
pref-607515-1rnwv   1/1    Running  0         4m   10.47.0.1  node2.k8s
pref-607515-27wp0   1/1    Running  0         4m   10.44.0.8  node1.k8s
pref-607515-5xd0z   1/1    Running  0         4m   10.44.0.5  node1.k8s
pref-607515-jx9wt   1/1    Running  0         4m   10.44.0.4  node1.k8s
pref-607515-mlgqm   1/1    Running  0         4m   10.44.0.6  node1.k8s
```

Из пяти созданных модулей четыре оказались в узле `node1`, и только один оказался в `node2`. Почему один из них оказался в узле `node2`, а не в узле `node1`? Причина этого в том, что, для того чтобы решить, куда назначить модуль, планировщик, помимо функции приоретизации сходства узлов, также использует ряд других функций. Одной из них является функция `SelectorSpreadPriority`, которая гарантирует, что модули, принадлежащие одному и тому же набору реплик ReplicaSet или службе, были распределены по разным узлам, поэтому аварийный сбой узла не приведет к аварийному сбою всей службы. Этот факт является наиболее вероятной причиной, почему один из модулей был назначен узлу `node2`.

Вы можете попробовать промасштабировать развертывание до 20 модулей или более, и вы увидите, что большинство модулей будет назначено узлу `node1`. В моем тесте только два из 20 были назначены узлу `node2`. Если бы вы не

определили предпочтения сходства узлов, то модули были бы распределены по двум узлам равномерно.

16.3 Совместное размещение модулей с использованием сходства и антисходства модулей

Вы увидели, как правила сходства узлов используются для влияния на то, какому узлу модуль назначается. Однако эти правила влияют только на сходство между модулем и узлом, в то время как иногда вы хотели бы иметь возможность указывать сходство между самими модулями.

Например, представьте, что у вас есть фронтенд- и бэкенд-модули. Развертывание этих модулей рядом друг с другом снижает задержку и повышает производительность приложения. Вы могли бы применить правила сходства узлов, чтобы обеспечить развертывание обоих на одном узле, стойке сервера или центре обработки данных, но тогда вам пришлось бы точно указывать, какому узлу, стойке или центру обработки данных их приписывать, что является не самым лучшим решением. Уж лучше дать системе Kubernetes разворачивать модули там, где она считает нужным, держа фронтенд- и бэкенд-модули близко друг к другу. Это может быть достигнуто с помощью *сходства модулей*. Давайте познакомимся с ним поближе на примере.

16.3.1 Использование межмодульного сходства для развертывания модулей на одном узле

Вы развернете бэкенд-модуль и пять реплик фронтенд-модуля со сходством модулей сконфигурированными так, чтобы все они были развернуты на том же узле, что и внутренний модуль.

Сначала разверните бэкенд-модуль:

```
$ kubectl run backend -l app=backend --image busybox -- sleep 999999
deployment "backend" created
```

Это развертывание никоим образом не является особенным. Единственное, что вам нужно отметить, – это метка `app=backend`, которую вы добавили в модуль с помощью параметра `-l`. Именно эту метку вы будете использовать в конфигурации `podAffinity` фронтенд-модуля.

Указание сходства модулей в определении модуля

Определение фронтенд-модуля показано в следующем ниже листинге.

Листинг 16.13. Модуль с использованием podAffinity: frontend-podaffinity-host.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
```

```
metadata:
  name: frontend
spec:
  replicas: 5
  template:
    ...
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                app: backend
    ...
```

Определение правил podAffinity

Определение жесткого требования, не предпочтения

Модули этого развертывания должны быть развернуты на том же узле, что и модули, которые совпадают с селектором

Данный листинг показывает, что такое развертывание создаст модули, которые имеют жесткое требование развертывания на одном узле (указанное в поле `topologyKey`), как модули, которые имеют метку `app=backend` (см. рис. 16.4).

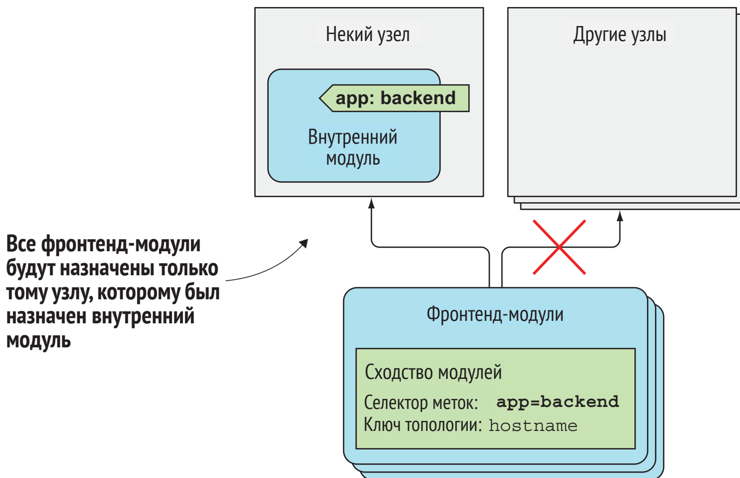


Рис. 16.4. Сходство модулей позволяет назначать модули узлу, где есть другие модули с определенной меткой

ПРИМЕЧАНИЕ. Вместо поля `matchLabels` вы также можете использовать более выразительное поле `matchExpressions`.

Развертывание модуля со сходством модулей

Прежде чем создавать это развертывание, давайте посмотрим, какому узлу был назначен бэкенд-модуль:

```
$ kubectl get po -o wide
NAME                 READY  STATUS   RESTARTS  AGE  IP         NODE
backend-257820-qhqj6 1/1    Running  0         8m   10.47.0.1  node2.k8s
```

При создании фронтенд-модулей они также должны быть развернуты на узле node2. Вы создадите развертывание и увидите, где модули развернуты. Это показано в следующем ниже листинге.

Листинг 16.14. Развертывание фронтенд-модулей и просмотр узла, которому они были назначены

```
$ kubectl create -f frontend-podaffinity-host.yaml
deployment «frontend» created
$ kubectl get po -o wide
NAME                  READY  STATUS   RESTARTS  AGE  IP         NODE
backend-257820-qhqj6  1/1    Running  0         8m   10.47.0.1  node2.k8s
frontend-121895-2c1ts 1/1    Running  0         13s  10.47.0.6  node2.k8s
frontend-121895-776m7 1/1    Running  0         13s  10.47.0.4  node2.k8s
frontend-121895-7ffsm 1/1    Running  0         13s  10.47.0.8  node2.k8s
frontend-121895-fpgm6 1/1    Running  0         13s  10.47.0.7  node2.k8s
frontend-121895-vb9ll 1/1    Running  0         13s  10.47.0.5  node2.k8s
```

Все фронтенд-модули действительно были назначены тому же узлу, что и бэкенд-модуль. При назначении фронтенд-модуля планировщик сначала нашел все модули, которые соответствуют селектору labelSelector, определенному в конфигурации правила podAffinity фронтенд-модуля, а затем назначил фронтенд-модуль тому же узлу.

Как планировщик использует правила сходства модулей

Что интересно, если вы сейчас удалите бэкенд-модуль, то планировщик назначит модуль узлу node2, даже если в нем не установлено никаких правил сходства модулей (эти правила находятся только на фронтенд-модулях). Это имеет смысл, потому что в противном случае, если бэкенд-модуль будет случайно удален и переназначен другому узлу, правила сходства фронтенд-модулей будут нарушены.

Если вы увеличите уровень детализации журналирования планировщика, а затем проверите его журнал, то вы можете убедиться, что планировщик принимает во внимание другие правила сходства модулей. В следующем далее листинге показаны соответствующие строки журнала.

Листинг 16.15. Журнал планировщика, показывающий, почему внутренний модуль назначается узлу node2

```
... Attempting to schedule pod: default/backend-257820-qhqj6
... ...
... backend-qhqj6 -> node2.k8s: Taint Toleration Priority, Score: (10)
```

```
... backend-qhqj6 -> node1.k8s: Taint Toleration Priority, Score: (10)
... backend-qhqj6 -> node2.k8s: InterPodAffinityPriority, Score: (10)
... backend-qhqj6 -> node1.k8s: InterPodAffinityPriority, Score: (0)
... backend-qhqj6 -> node2.k8s: SelectorSpreadPriority, Score: (10)
... backend-qhqj6 -> node1.k8s: SelectorSpreadPriority, Score: (10)
... backend-qhqj6 -> node2.k8s: NodeAffinityPriority, Score: (0)
... backend-qhqj6 -> node1.k8s: NodeAffinityPriority, Score: (0)
... Host node2.k8s => Score 100030
... Host node1.k8s => Score 100022
... Attempting to bind backend-257820-qhqj6 to node2.k8s
```

Если вы обратите внимание на две строки, выделенные жирным шрифтом, то увидите, что во время назначения бэкенд-модуля узел node2 получил более высокую оценку, чем узел node1, из-за межмодульного сходства.

16.3.2 Развертывание модулей в одной стойке, зоне доступности или географическом регионе

В предыдущем примере вы использовали правило podAffinity для развертывания фронтенд-модулей на том же узле, что и бэкенд-модули. Вы, вероятно, не хотите, чтобы все ваши фронтенд-модули работали на одной и той же машине, но все равно хотели бы держать их рядом с бэкенд-модулем – например, чтобы запускать их в одной и той же зоне доступности.

Совместное размещение модулей в одной зоне доступности

Кластер, который я использую, работает в трех виртуальных машинах на моей локальной машине, поэтому все узлы находятся, так сказать, в одной зоне доступности. Но если бы узлы были в разных зонах, то, для того чтобы запустить фронтенд-модули в той же зоне, что и внутренний модуль, мне пришлось бы только поменять значение свойства topologyKey на failure-domain.beta.kubernetes.io/zone.

Совместное размещение модулей в одном географическом регионе

Для того чтобы разрешить развертывание модулей в одном и том же регионе вместо одной и той же зоны (центры обработки данных поставщиков облачных служб обычно расположены в разных географических регионах и разделены на несколько зон доступности в каждом регионе), для параметра topologyKey будет задано значение failure-domain.beta.kubernetes.io/region.

Как работает ключ topologyKey

Принцип работы ключа topologyKey очень прост. Три ключа, которых мы коснулись до этого, не являются специальными. Если вы хотите, вы можете легко использовать свой собственный ключ topologyKey, такой как стойка rack, с тем чтобы модули назначались одной и той же стойке сервера. Единст-

венным предварительным условием является добавление метки `rack` в узлы. Этот сценарий показан на рис. 16.5.

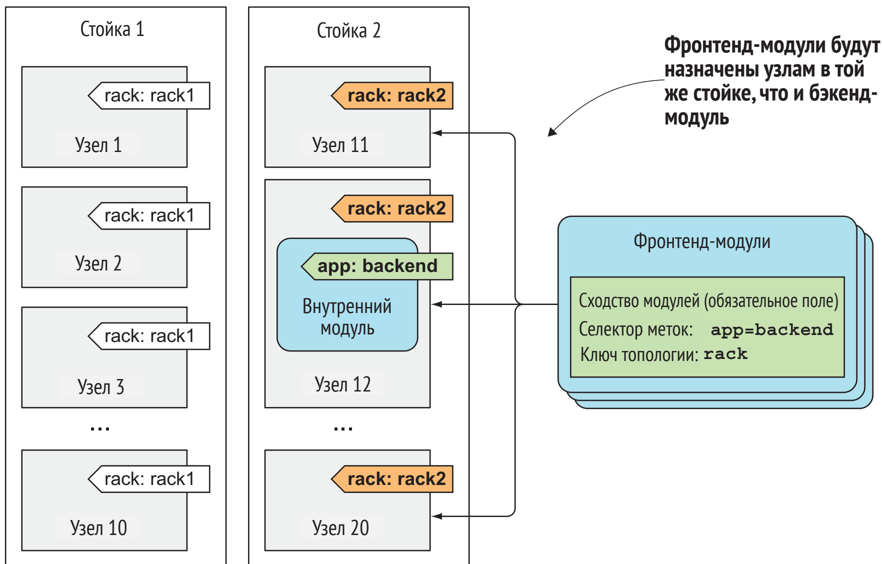


Рис. 16.5. Ключ топологии topologyKey в правиле podAffinity определяет область действия, куда должен быть назначен модуль

Например, если у вас было 20 узлов, по 10 в каждой стойке, то вы пометите первые десять как `rack=rack1` и другие как `rack=rack2`. Затем при определении правила `podAffinity` модуля вы задаете ключу `toplogyKey` значение `rack`.

Когда планировщик решает, где развернуть модуль, он проверяет конфигурацию сходства модулей, находит модули, которые соответствуют селектору меток, и ищет узлы, на которых они работают. В частности, он ищет метку узлов, ключ которой соответствует полю `topologyKey`, указанному в правиле `podAffinity`. Затем он выбирает все узлы, метка которых соответствует значениям модулей, которые он нашел ранее. На рис. 16.5 селектор меток совпал с внутренним модулем, который работает на узле 12. Значение метки `rack` на этом узле равно `rack2`, поэтому при назначении фронтенд-модуля планировщик будет выбирать только среди узлов, имеющих метку `rack=rack2`.

> **ПРИМЕЧАНИЕ.** По умолчанию селектор меток отождествляет модули только в том же пространстве имен, что и назначаемый модуль. Но вы также можете выбирать модули из других пространств имен, добавив поле `namespaces` на том же уровне, что и селектор `labelSelector`.

16.3.3 Выражение предпочтений сходства модулей вместо жестких требований

Ранее, когда мы говорили о сходстве узлов, вы видели, что правило `nodeAffinity` может использоваться для выражения жесткого требования,

имея в виду, что модуль назначается только тем узлам, которые соответствуют правилам сходства узлов. Оно также может использоваться для указания предпочтительности тех или иных узлов, чтобы поручить планировщику назначать модуль определенным узлам, позволяя ему назначать модуль в любом другом месте, если по какой-либо причине эти узлы не могут уместить данный модуль.

То же самое относится и к правилу `podAffinity`. Вы можете сообщить планировщику, что предпочитаете, чтобы ваши фронтенд-модули были назначены тому же узлу, что и ваш бэкенд-модуль, но если это невозможно, то вы согласны с тем, что они будут назначены в другом месте. Пример развертывания с использованием правила сходства модулей `preferredDuringSchedulingIgnoredDuringExecution` показан в следующем ниже листинге.

Листинг 16.16. Предпочтение сходства модулей

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 5
  template:
    ...
    spec:
      affinity:
        podAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:    <- Предпочитаемый вместо требуемый
          - weight: 80
            podAffinityTerm:
              topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  app: backend
      containers: ...
```

Компонента веса и сходства модулей указана как в предыдущем примере

Как и в правилах предпочтений `nodeAffinity`, для каждого правила необходимо определить вес. Также нужно указать `topologyKey` и `labelSelector`, как в правилах жесткого требования `podAffinity`. На рис. 16.6 показан этот сценарий.

Развертывание этого модуля, как и в примере с правилом `nodeAffinity`, развертывает четыре модуля на том же узле, что и бэкенд-модуль, и один модуль на другом узле (см. следующий ниже листинг).

Листинг 16.17. Модули, развернутые с предпочтениями сходства модулей

```
$ kubectl get po -o wide
NAME                   READY  STATUS   RESTARTS  AGE  IP         NODE
backend-257820-ssrgj   1/1    Running  0         1h   10.47.0.9  node2.k8s
```

```
frontend-941083-3mff9  1/1    Running  0         8m   10.44.0.4  node1.k8s
frontend-941083-7fp7d  1/1    Running  0         8m   10.47.0.6  node2.k8s
frontend-941083-cq23b  1/1    Running  0         8m   10.47.0.1  node2.k8s
frontend-941083-m70sw  1/1    Running  0         8m   10.47.0.5  node2.k8s
frontend-941083-wsjv8  1/1    Running  0         8m   10.47.0.4  node2.k8s
```

16.3.4 Назначение модулей на удалении друг от друга с помощью антисходства модулей

Вы увидели, как сообщать планировщику, чтобы модули размещались совместно, но иногда вам может потребоваться прямо противоположное. Вы можете удерживать модули на удалении друг от друга. Это называется антисходством модулей. Оно определяется таким же образом, как сходство модулей, за исключением того, что вместо свойства `podAffinity` вы используете свойство `podAntiAffinity`, которое заставляет планировщика никогда не выбирать узлы, где работают модули, которые соответствуют селектору меток свойства `podAntiAffinity`, как показано на рис. 16.7.

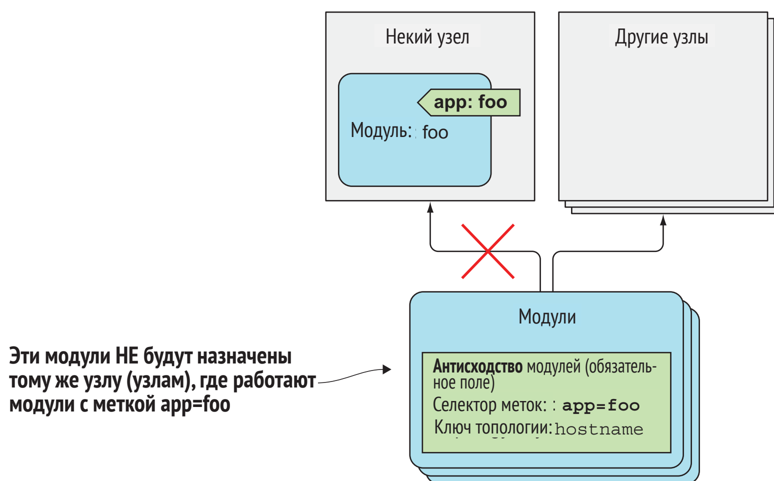


Рис. 16.7. Использование антисходства модулей, для того чтобы держать модули на удалении от узлов, которое выполняют модули с определенной меткой

Пример того, почему вам может потребоваться использовать антисходство модулей, встречается, когда два набора модулей мешают производительности друг друга, в случае если они работают на одном узле. В этом случае вы хотите сообщить планировщику никогда не назначать эти модули на одном узле. Еще один пример – заставить планировщик распределять модули одной и той же группы по разным зонам доступности или регионам, чтобы аварийный сбой всей зоны (или региона) никогда не приводил к полному отключению службы.

Использование антисходства для разделения модулей одного развертывания

Давайте посмотрим, как заставить назначать ваши фронтенд-модули разным узлам. Следующий ниже листинг показывает, как конфигурируется антисходство модулей.

Листинг 16.18. Модули с антисходством: frontend-podantiaffinity-host.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: frontend
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                app: frontend
      containers: ...
```

Фронтенд-модули имеют метку app=frontend

Определение жестких требований к антисходству модулей

Фронтенд-модуль должен быть назначен той же машине, что и модуль с меткой app=frontend

На этот раз вместо свойства `podAffinity` вы определяете свойство `podAntiAffinity`, и вы заставляете селектор `labelSelector` отождествлять те же модули, которые создаются размещением. Давайте посмотрим, что произойдет при создании этого развертывания. Созданные им модули показаны в следующем ниже листинге.

Листинг 16.19. Модули, созданные развертыванием

```
$ kubectl get po -l app=frontend -o wide
NAME                  READY  STATUS   RESTARTS  AGE  IP         NODE
frontend-286632-0lffz 0/1    Pending  0         1m   <none>
frontend-286632-2rkcz 1/1    Running  0         1m   10.47.0.1  node2.k8s
frontend-286632-4nwhp 0/1    Pending  0         1m   <none>
frontend-286632-h4686 0/1    Pending  0         1m   <none>
frontend-286632-st222 1/1    Running  0         1m   10.44.0.4  node1.k8s
```

Как вы можете видеть, только два модуля были назначены: один – узлу `node1`, другой – узлу `node2`. Три оставшихся модуля находятся в ожидании, потому что планировщик не может их назначить тем же узлам.

Использование предпочтительного антисходства модулей

В этом случае, скорее всего, вам следовало бы указать мягкое требование (используя свойство `preferredDuringSchedulingIgnoredDuringExecution`). В конце концов, не такая уж большая проблема в том, если два фронтенд-модуля работают на одном узле. Но в сценариях, где это представляет проблему, использование свойства `requiredDuringScheduling` будет самым подходящим.

Как и со сходством модулей, свойство `topologyKey` определяет область действия, где модуль не должен быть развернут. Его можно использовать, чтобы гарантировать, что модули не будут развернуты в той же стойке, зоне доступности, регионе или любой индивидуально настроенной области действия, созданной с помощью собственных меток узлов.

16.4 Резюме

В этой главе мы рассмотрели, как гарантировать, чтобы модули не назначались определенным узлам или назначались только определенным узлам либо из-за меток узлов, либо из-за работающих на них модулей.

Вы узнали, что:

- если добавить в узел ограничение, то модули не будут назначаться этому узлу, в случае если они не будут допускать наличие этого дефекта;
- существует три типа ограничений: `NoSchedule` полностью предотвращает назначение модуля узлу, `PreferNoSchedule` не такое строгое, как предыдущее, и `NoExecute`, которое даже вытесняет существующие модули из узла;
- ограничение `NoExecute` также используется для указания того, как долго плоскость управления должна ждать, прежде чем переназначить модуль, когда узел, на котором он работает, становится недостижимым или неготовым;
- сходство узлов позволяет указывать то, каким узлам должен быть назначен модуль. Его можно использовать для определения жесткого требования или только для выражения предпочтительности узла;
- сходство модулей используется для того, чтобы планировщик развертывал модули на том же узле, где работает другой модуль (на основе меток модуля);
- свойство `topologyKey` сходства модуля указывает на то, как близко модуль должен быть развернут от другого модуля (на том же узле или на узле в той же стойке, зоне доступности или регионе доступности);
- антисходство модуля можно использовать для того, чтобы держать некоторые модули на удалении друг от друга;
- как сходство модуля, так и антисходство, в частности сходство узлов, могут указывать на жесткие требования или предпочтения.

В следующей главе вы познакомитесь с рекомендациями по разработке приложений и с тем, как обеспечивать их бесперебойную работу в среде Kubernetes.

Глава 17

Рекомендации по разработке приложений

Эта глава посвящена:

- характеристике ресурсов Kubernetes, которые появляются в типичном приложении;
- добавлению постстартовых и предостановочных обработчиков жизненного цикла модуля;
- правильному завершению работы приложения без нарушения запросов клиентов;
- упрощению управления приложениями в Kubernetes;
- использованию контейнеров инициализации в модуле;
- локальной разработке с помощью Minikube.

К настоящему моменту мы рассмотрели подавляющую часть того, что вам нужно знать для запуска приложений в Kubernetes. Мы познакомились с тем, что делает каждый отдельный ресурс и как он используется. Теперь мы увидим, как объединять их в типичном приложении, работающем на Kubernetes. Мы также рассмотрим, как делать так, чтобы приложение работало гладко. Ведь в этом весь смысл использования Kubernetes, не правда ли?

Надеюсь, эта глава поможет прояснить любые недоразумения и объяснить все то, что еще не было четко разъяснено. Попутно мы также представим несколько дополнительных концепций, которые до сих пор не упоминались.

17.1 Соединение всего вместе

Давайте начнем с того, что рассмотрим, из чего состоит настоящее приложение. Это также даст вам возможность убедиться в том, что вы помните все, что вы узнали к этому моменту, и посмотреть на общую картину. На рис. 17.1 показаны компоненты Kubernetes, используемые в типичном приложении.

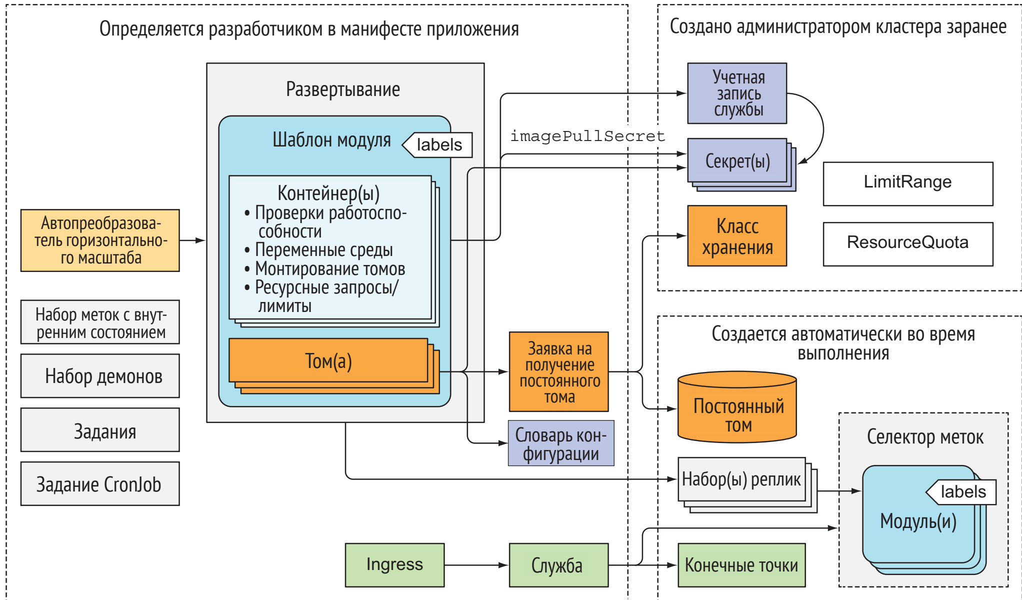


Рис. 17.1. Ресурсы в типичном приложении

Типичный манифест приложения содержит один или несколько объектов Deployment и/или StatefulSet. К ним относятся шаблон модуля, содержащий один или несколько контейнеров с проверкой живучести для каждого из них и проверкой готовности служб, доступ к которым предоставляется контейнером (если таковые имеются). К модулям, предоставляющим службы другим пользователям, обеспечивается доступ через одну или несколько служб Service. Если они должны быть достижимы за пределами кластера, то эти службы либо конфигурируются как службы `LoadBalancer` или службы типа `NodePort`, либо предоставляются через ресурс Ingress.

Шаблоны модуля (и создаваемые из них модули) обычно ссылаются на два типа секретов – для извлечения образов контейнеров из приватных хранилищ образов и те, которые используются непосредственно процессом, работающим внутри модулей. Сами секреты обычно не являются частью манифеста приложения, так как они конфигурируются не разработчиками приложения, а системными администраторами. Секреты обычно назначаются учетным записям ServiceAccount, которые назначаются отдельным модулям.

Приложение также содержит один или более словарей конфигурации ConfigMap, которые используются для инициализации переменных среды или монтируются в качестве тома `configMap` в модуле. Некоторые модули используют дополнительные тома, в частности `emptyDir` или `gitRepo`, тогда как модули, требующие постоянного хранения, используют тома на основе заявки persistentVolumeClaim. Тома PersistentVolumeClaim также являются частью манифеста приложения, тогда как классы хранения StorageClass, на которые они ссылаются, создаются системными администраторами заранее.

В некоторых случаях приложение также требует использования задания Job или CronJob. Наборы демонов DaemonSet обычно не являются частью развертываний приложений, но обычно создаются системными администраторами для запуска системных служб на всех или на подмножестве узлов. Автопреобразователи масштаба HorizontalPodAutoscaler либо включаются в манифест разработчиками, либо добавляются позже системными администраторами. Администратор кластера также создает объекты LimitRange и ResourceQuota, чтобы контролировать использование вычислительных ресурсов отдельными модулями и всеми модулями (в целом).

После развертывания приложения различные контроллеры Kubernetes автоматически создают дополнительные объекты. К ним относятся объекты Endpoints, создаваемые контроллером конечных точек, наборы реплик ReplicaSet, создаваемые контроллером развертывания, и фактические модули, создаваемые контроллерами набора реплик ReplicaSet (либо задания Job, задания CronJob, набора StatefulSet или набора DaemonSet).

Ресурсы часто помечаются одной или несколькими метками, чтобы держать их организованными. Это относится не только к модулям, но и ко всем другим ресурсам. В дополнение к меткам многие ресурсы также содержат аннотации, описывающие каждый ресурс, содержат контактную информацию ответственного за него лица или группы либо предоставляют дополнительные метаданные для управления и другие инструменты.

В центре всего этого находится объект Pod, модуль, который, возможно, является самым важным ресурсом Kubernetes. В конце концов, каждое из ваших приложений работает внутри него. Для того чтобы убедиться, что вы знаете, как разрабатывать приложения, которые максимально используют свою среду, давайте в последний раз внимательно рассмотрим модули – на этот раз с точки зрения приложения.

17.2 Жизненный цикл модуля

Мы отметили, что модули можно сравнить с виртуальными машинами, предназначенными для работы только одного приложения. Хотя приложение, запущенное в модуле, не отличается от приложения, запущенного в виртуальной машине, между ними существуют значительные различия. Одним из примеров является то, что приложения, работающие в модуле, могут быть уничтожены в любое время, потому что система Kubernetes должна переместить модуль на другой узел по причине или из-за запроса на уменьшение масштаба. Мы рассмотрим этот аспект далее.

17.2.1 Приложения должны ожидать, что они могут быть удалены и перемещены

За пределами Kubernetes работающие в виртуальных машинах приложения редко перемещаются с одной машины на другую. Когда оператор перемещает приложение, он также может перенастроить приложение и вручную

проверить, что в новом местоположении приложение работает нормально. В Kubernetes приложения перемещаются гораздо чаще и автоматически – никакой человек-оператор их не перенастраивает и не убеждается, что после перемещения они по-прежнему работают как нужно. Это означает, что разработчики приложений должны стараться делать так, чтобы их приложения допускали относительно частые перемещения.

Приложения должны ожидать, что IP-адрес и хостнейм могут быть изменены

Когда модуль уничтожается и запускается в другом месте (технически это новый экземпляр модуля, заменяющий старый; модуль не перемещается), у него не только новый IP-адрес, но и новое имя и хостнейм. Большинство приложений без внутреннего состояния, как правило, может справиться с этим без каких-либо побочных эффектов, но приложения с внутренним состоянием, как правило, не могут. Мы узнали, что приложения с внутренним состоянием можно запускать через объект StatefulSet, который гарантирует, что при запуске приложения на новом узле, после того как он переназначен, он по-прежнему будет видеть тот же хостнейм и постоянное состояние, как и раньше. Тем не менее IP-адрес модуля поменяется. Приложения должны быть к этому готовы. Поэтому разработчик приложения никогда не должен основывать принадлежность в кластеризованном приложении на IP-адресе участника, а если и основывать его на хостнейме, то всегда следует использовать объект StatefulSet.

Приложения должны ожидать, что данные, записанные на диск, исчезнут

Еще один аспект, который следует учитывать, заключается в том, что если приложение записывает данные на диск, то при условии неподключения постоянного хранилища в месте, в которое приложение выполняет запись, эти данные могут стать недоступными после запуска приложения в новом модуле. Вполне понятно, что это происходит, когда модуль переназначается, но файлы, записанные на диск, исчезнут даже в сценариях, которые не включают переназначение. Даже в течение жизни одного модуля файлы, записанные на диск приложением, работающим в модуле, могут исчезнуть. Это стоит объяснить на примере.

Представьте себе приложение, которое имеет длинную и вычислительно емкую процедуру начального запуска. Чтобы помочь приложению работать быстрее при последующих запусках, разработчики кешируют результаты начального запуска на диске (примером этого может быть сканирование всех классов Java в поисках аннотаций при запуске, а затем запись результатов в индексный файл). Поскольку приложения в Kubernetes по умолчанию выполняются в контейнерах, эти файлы записываются в файловую систему контейнера. Если контейнер затем перезапускается, то все они будут потеряны, потому что новый контейнер начинается с совершенно нового доступного для записи слоя (см. рис. 17.2).

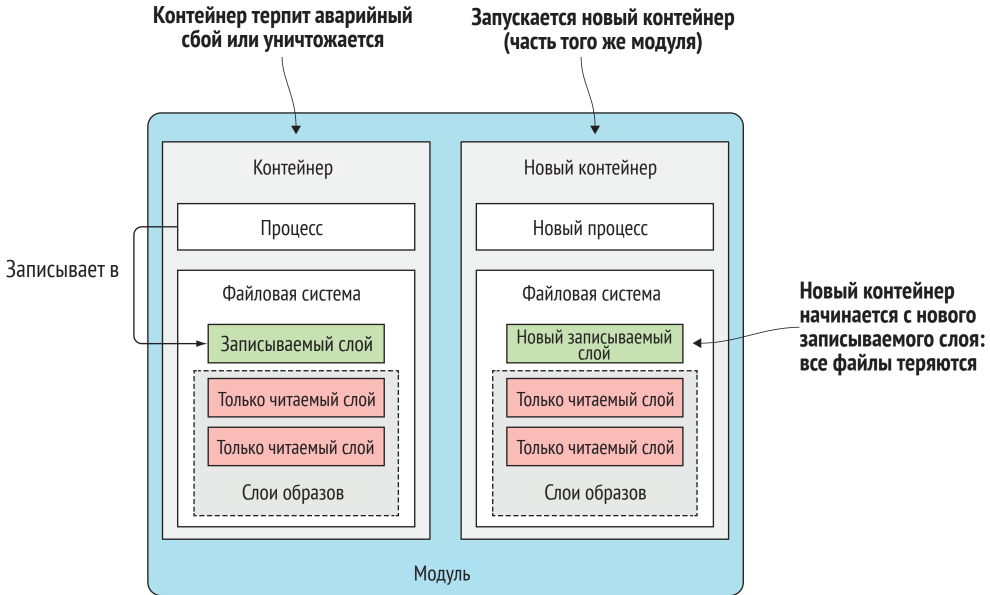


Рис. 17.2. Файлы, записанные в файловой системе контейнера, теряются, когда контейнер перезапускается

Не забывайте, что отдельные контейнеры могут быть перезапущены по нескольким причинам, например из-за фатального сбоя процесса, из-за того, что проверка живучести вернула несработку, или из-за того, что у узла закончилась память и процесс был уничтожен убийцей OOMKiller. Когда это случается, модуль остается прежним, но вот сам контейнер будет совершенно новым. Агент Kubelet не запускает один и тот же контейнер снова; он всегда создает новый контейнер.

Использование томов для сохранения данных при перезапусках контейнера

При перезапуске его контейнера приложению в примере потребуется выполнить сложную процедуру запуска еще раз. Это может быть или не быть желательным. Для того чтобы такие данные не были потеряны, необходимо использовать, по крайней мере, том с областью действия модуля. Поскольку тома живут и умирают вместе с модулем, новый контейнер будет иметь возможность повторно использовать данные, записанные в такой том предыдущим контейнером (рис. 17.3).

Использование тома для сохранения файлов при перезапусках контейнера иногда является отличной идеей, но не всегда. Что делать, если данные будут повреждены и снова будут приводить вновь созданный процесс к аварийному завершению работы? Это приведет к непрерывному циклу сбоя (модуль будет показывать статус `CrashLoopBackOff`). Если бы вы не использовали том, то новый контейнер начинался бы с нуля и, скорее всего, не имел бы фатального сбоя. Использование томов для сохранения файлов при перезапусках контей-

нера является палкой о двух концах. И нужно хорошенько подумать, использовать их или нет.

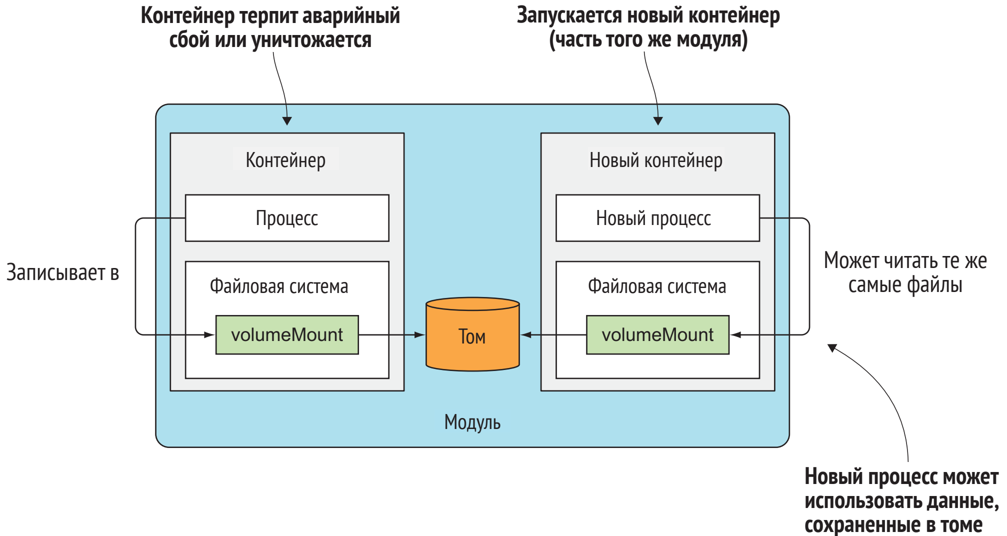


Рис. 17.3. Использование тома для сохранения данных при перезапусках контейнера

17.2.2 Переназначение мертвых или частично мертвых модулей

Если контейнер модуля продолжает сбоить, то агент Kubelet будет бесконечно его перезапускать. Время между перезапусками будет увеличиваться экспоненциально, пока не достигнет пяти минут. Во время этих пятиминутных интервалов модуль, по существу, будет мертвым, потому что процесс его контейнера не работает. По правде говоря, если это мультиконтейнерный модуль, то некоторые контейнеры могут работать нормально, поэтому модуль только частично будет мертвым. Но если модуль содержит всего один-единственный контейнер, то модуль будет практически мертвым и совершенно бесполезным, потому что никакой процесс в нем больше не работает.

Вы можете удивиться, узнав, что такие модули не удаляются и не переназначаются автоматически, даже если они являются частью набора реплик ReplicaSet или аналогичного контроллера. Если вы создадите реплику с требуемым числом желаемых реплик, равным трем, а затем один из контейнеров в одном из этих модулей начнет сбоить, то Kubernetes не удалит и не заменит модуль. В результате этого набор ReplicaSet будет только с двумя правильно работающими репликами вместо желаемых трех (рис. 17.4).

Вероятно, вы ожидаете, что модуль будет удален и заменен другим экземпляром модуля, который может успешно работать на другом узле. В конце концов, контейнер может аварийно завершить работу из-за проблемы, связанной с узлом, которая не проявляется на других узлах. К сожалению, это не

так. Контроллеру набора ReplicaSet все равно, мертвы модули или нет, – его интересует только то, чтобы количество модулей соответствовало требуемому количеству реплик, что в данном случае и происходит.

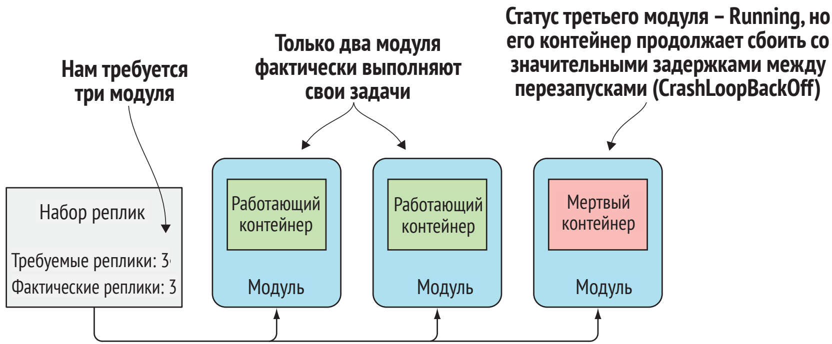


Рис. 17.4. Контроллер набора ReplicaSet не переназначает мертвые модули

Если вы хотите убедиться сами, я включил манифест YAML для набора реплик, чьи модули будут продолжать давать сбои (см. файл replicaset-crashingpods.yaml в архиве кода). Если вы создадите набор реплик и проинспектируете созданные модули, то увидите следующий ниже листинг.

Листинг 17.1. Набор ReplicaSet и модули, которые продолжают давать аварийный сбой

```
$ kubectl get po
NAME                 READY  STATUS            RESTARTS  AGE
crashing-pods-f1tcd  0/1    CrashLoopBackOff  5         6m
crashing-pods-k7l6k  0/1    CrashLoopBackOff  5         6m
crashing-pods-z7l3v  0/1    CrashLoopBackOff  5         6m

$ kubectl describe rs crashing-pods
Name:           crashing-pods
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed

$ kubectl describe po crashing-pods-f1tcd
Name:           crashing-pods-f1tcd
Namespace:      default
Node:           minikube/192.168.99.102
Start Time:     Thu, 02 Mar 2017 14:02:23 +0100
Labels:         app=crashing-pods
Status:         Running
```


До известной степени вполне понятно, почему Kubernetes ведет себя таким образом. Контейнер будет перезапускаться каждые пять минут в надежде, что основная причина аварийного сбоя будет устранена. Логическое обоснование

этому заключается в том, что переназначение модуля другому узлу, скорее всего, не устранит проблему в любом случае, потому что приложение работает внутри контейнера, и все узлы должны быть в основном эквивалентны. Это не всегда так, но в большинстве случаев так и случается.

17.2.3 Запуск модулей в определенном порядке

Другое различие между приложениями, работающими в модулях, и управляемыми вручную состоит в том, что специалист группы системного администрирования, который развертывает эти приложения, знает о зависимостях между ними. И это позволяет ему запускать приложения по порядку.

Как запускаются модули

Когда вы используете Kubernetes для запуска приложений с несколькими модулями, у вас нет готового способа сообщить системе Kubernetes сначала запускать определенные модули, а остальные только тогда, когда первые модули уже подняты и готовы обслуживать. Безусловно, вы можете отправить манифест для первого приложения, а затем дождаться готовности модуля (модулей), перед тем как отправить второй манифест, но вся ваша система обычно определяется в одном файле YAML или JSON, содержащем несколько модулей, служб и других объектов.

Сервер API Kubernetes обрабатывает объекты в YAML/JSON в том порядке, в котором они перечислены, но это только означает, что они записываются в хранилище etcd в этом порядке. У вас не будет никакой гарантии, что модули будут запускаться в таком порядке.

Однако вы *можете* предотвращать запуск главного контейнера модуля до тех пор, пока не будет выполнено предварительное условие. Это делается путем включения в модуль контейнеров инициализации.

Знакомство с контейнерами инициализации

В дополнение к регулярным контейнерам модули могут также включить контейнеры инициализации (init). Как следует из названия, их можно использовать для инициализации модуля – это часто означает запись данных в тома модуля, которые затем монтируются в главный контейнер модуля.

Модуль может иметь любое количество контейнеров инициализации. Они выполняются последовательно, и только после того, как последний завершит работу, будут запущены главные контейнеры модуля. Это означает, что контейнеры инициализации также могут использоваться для задержки запуска главных контейнеров модуля, например до выполнения определенного предварительного условия. Контейнер инициализации может дожидаться, когда будет поднята и готова к работе служба, требуемая главным контейнером модуля. Когда это наступает, контейнер инициализации завершает свою работу и позволяет запустить главный контейнер (контейнеры). Благодаря этому главный контейнер не будет использовать службу до того, как она станет готовой.

Рассмотрим пример модуля, использующего контейнер инициализации для задержки запуска главного контейнера. Помните модуль `fortune`, который вы создали в главе 7? Это веб-сервер, который в качестве отклика на запросы клиентов возвращает цитату. Теперь давайте представим, что у вас есть модуль `fortune-client`, который требует, чтобы служба `fortune` была запущена до запуска ее главного контейнера. Вы можете добавить контейнер инициализации, который проверяет, откликается служба на запросы или нет. До тех пор, пока это не произойдет, контейнер инициализации продолжает повторять попытки. Как только он получает отклик, контейнер инициализации завершается и дает запуститься главному контейнеру.

Добавление контейнера инициализации в модуль

Контейнеры инициализации могут быть определены в спецификации модуля так же, как главные контейнеры, но посредством поля `spec.initContainers`. Вы найдете полный YAML для модуля `fortune-client` в архиве кода, прилагаемого к этой книге. В следующем ниже листинге показан фрагмент, в котором определяется контейнер инициализации.

Листинг 17.2. Контейнер инициализации, определенный в модуле: fortune-client.yaml

```
spec:
  initContainers:
  - name: init
    image: busybox
    command:
    - sh
    - -c
    - 'while true; do echo "Waiting for fortune service to come up...";
    ➥ wget http://fortune -q -T 1 -O /dev/null >/dev/null 2>/dev/null
    ➥ && break; sleep 1; done; echo "Service is up! Starting main
    ➥ container."'
```

Вы определяете контейнер инициализации, не обычный контейнер

Контейнер инициализации выполняет цикл, который работает до тех пор, пока служба fortune не будет поднята

При развертывании этого модуля запускается только его контейнер инициализации. Это будет показано в статусе модуля, когда вы выведете список модулей с помощью команды `kubectl get`:

```
$ kubectl get po
NAME             READY   STATUS     RESTARTS   AGE
fortune-client   0/1     Init:0/1   0          1m
```

В столбце `STATUS` показано, что завершен нуль из одного контейнера инициализации. Вы можете увидеть журнал контейнера инициализации с помощью команды `kubectl logs`:

```
$ kubectl logs fortune-client -c init
Waiting for fortune service to come up...
```

При выполнении команды `kubectl logs` вам нужно указать имя контейнера инициализации вместе с переключателем `-c` (в данном примере именем контейнера инициализации модуля является `init`, как показано в листинге 17.2).

Главный контейнер не будет работать до тех пор, пока вы не развернете службу `fortune` и модуль `fortune-server`. Вы найдете их в файле fortune-server.yaml.

Рекомендации по работе с межмодульными зависимостями

Вы видели, как контейнер инициализации можно использовать для задержки запуска главного контейнера(ов) модуля до тех пор, пока не будет выполнено предварительное условие (к примеру, обеспечивающее, что служба, от которой зависит модуль, будет готова), но гораздо лучше писать приложения, которые перед запуском приложения не требуют ждать готовности каждой службы, на которую они опираются. В конце концов, позже служба к тому же может перейти в автономный режим, в то время как приложение уже работает.

Приложение должно внутренне обрабатывать возможность того, что его зависимости не готовы. И не забывайте про проверки готовности. Если приложение не может выполнить свою работу, потому что одна из его зависимостей отсутствует, оно должно сигнализировать об этом через свою проверку готовности, поэтому система Kubernetes знает, что оно тоже не готово. Вы захотите сделать это не только потому, что это предотвращает добавление приложения в качестве конечной точки службы, но и потому, что готовность приложения также используется контроллером развертывания при выполнении скользящего обновления, тем самым предотвращая развертывание плохой версии.

17.2.4 Добавление обработчиков жизненного цикла

Мы говорили о том, как контейнеры инициализации можно использовать для подключения к запуску модуля, но модули также позволяют определять два обработчика жизненного цикла:

- постстартовый обработчик;
- предостановочный обработчик.

Такие обработчики жизненного цикла определяются для каждого контейнера, в отличие от контейнеров инициализации, которые применяются ко всему модулю. Как следует из их названий, они исполняются, когда контейнер запускается и перед тем, как он терминируется.

Обработчики жизненного цикла подобны проверкам живучести и готовности в том, что они могут:

- исполнять команду внутри контейнера;
- выполнять запрос HTTP GET по URL-адресу.

Давайте рассмотрим два обработчика по отдельности, чтобы увидеть, какое влияние они оказывают на жизненный цикл контейнера.

Использование постстартового обработчика жизненного цикла контейнера

Постстартовый обработчик выполняется сразу после запуска главного процесса контейнера. Он используется для выполнения дополнительных операций при запуске приложения. Разумеется, если вы являетесь автором приложения, которое работает в контейнере, вы всегда можете выполнять эти операции внутри самого кода приложения. Но когда вы запускаете приложение, разработанное кем-то другим, вы в основном не хотите (или не можете) изменять его исходный код. Постстартовые обработчики позволяют запускать дополнительные команды без необходимости касаться приложения. Они могут сигнализировать внешнему прослушивателю, что приложение запускается, либо они могут инициализировать приложение, чтобы оно могло начать выполнять свою работу.

Данный обработчик выполняется параллельно с главным процессом. Его имя может ввести в заблуждение, поскольку он не ожидает полного запуска главного процесса (если процесс имеет процедуру инициализации, то агент Kubelet, совершенно очевидно, не может дожидаться завершения процедуры, потому что у него нет никакого способа узнать, когда это произойдет).

Но даже если обработчик выполняется асинхронно, он влияет на контейнер двумя способами. До тех пор, пока обработчик не завершится, контейнер останется в состоянии ожидания `Waiting` с причиной `ContainerCreating`. И следовательно, статус модуля будет `Pending`, а не `Running`. Если обработчик не выполняется или возвращает ненулевой код выхода, то главный контейнер будет уничтожен.

Манифест модуля, содержащий постстартовый обработчик, выглядит следующим образом.

Листинг 17.3. Модуль с постстартовым обработчиком жизненного цикла: post-start-hook.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-poststart-hook
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    lifecycle:
      postStart:
        exec:
          command:
          - sh
          - -c
          - "echo 'hook will fail with exit code 15'; sleep 5; exit 15"
```

Обработчик исполняется, когда контейнер запускается

Он исполняет сценарий postStart.sh в каталоге /bin внутри контейнера

В данном примере команды `echo`, `sleep` и `exit` исполняются вместе с главным процессом контейнера, как только контейнер был создан. Вместо выполнения такой команды вы обычно выполняете шелл-скрипт или исполняемый файл, хранящийся в образе контейнера.

К сожалению, если процесс, начатый обработчиком, выводит журнал событий в стандартный выход, то вы нигде не сможете увидеть результаты. Это делает отладку обработчиков жизненного цикла неудобной. В случае сбоя обработчика вы увидите среди событий модуля только предупреждение `FailedPostStartHook` (вы можете увидеть их с помощью команды `kubectl describe pod`). Как показано в следующем ниже листинге, некоторое время спустя вы увидите более подробную информацию о том, почему произошел сбой обработчика.

Листинг 17.4. События модуля, показывающие код выхода неуспешного командного обработчика

```
FailedSync Error syncing pod, skipping: failed to "StartContainer" for
           "kubia" with PostStart handler: command 'sh -c echo 'hook
           will fail with exit code 15'; sleep 5 ; exit 15' exited
           with 15: : "PostStart Hook Failed"
```

Число 15 в последней строке является кодом выхода команды. При использовании обработчика HTTP GET причина может выглядеть следующим образом (вы можете попробовать это, развернув файл post-start-hook-httpget.yaml из архива кода, прилагаемого к этой книге).

Листинг 17.5. События модуля, показывающие причину сбоя обработчика HTTP GET

```
FailedSync Error syncing pod, skipping: failed to "StartContainer" for
           "kubia" with PostStart handler: Get
           http://10.32.0.2:9090/postStart: dial tcp 10.32.0.2:9090:
           getsockopt: connection refused: "PostStart Hook Failed"
```

> **ПРИМЕЧАНИЕ.** Постстартовый обработчик преднамеренно сконфигурирован неправильно и использует порт 9090 вместо правильного порта 8080, чтобы показать, что происходит, когда обработчик не срабатывает.

Стандартные и ошибочные результаты постстартовых командных обработчиков никуда не журналируются, поэтому вам, возможно, потребуется, чтобы процесс, который вызывает обработчик, писал в журнальный файл в файловой системе контейнера, что позволит вам исследовать содержимое файла примерно таким способом:

```
$ kubectl exec my-pod cat logfile.txt
```

Если контейнер по какой-либо причине перезапускается (в том числе из-за сбоя обработчика), то этот файл может быть удален, прежде чем вы сможе-

те его исследовать. Вы можете предпринять обходной маневр, смонтировав в контейнер том `emptyDir` и дав обработчику вести туда запись.

Использование предостановочного обработчика жизненного цикла контейнера

Предостановочный обработчик исполняется непосредственно перед тем, как контейнер терминируется. Когда контейнер должен быть терминирован, агент Kubelet выполнит предостановочный обработчик, если он сконфигурирован, и только затем отправит сигнал `SIGTERM` процессу (и позже уничтожит процесс, если он не завершится корректно).

Предостановочный обработчик может использоваться для того, чтобы инициировать корректное выключение контейнера, если он не выключается корректно по получении сигнала `SIGTERM`. Они также могут использоваться для выполнения произвольных операций перед завершением работы без необходимости реализации этих операций в самом приложении (это полезно, когда вы запускаете стороннее приложение, к исходному коду которого у вас нет доступа и/или который вы не можете модифицировать).

Конфигурирование предостановочного обработчика в манифесте модуля не сильно отличается от добавления постстартового обработчика. В предыдущем примере показан постстартовый обработчик, который исполняет команду, поэтому теперь мы рассмотрим предостановочный обработчик, который выполняет запрос HTTP GET. В следующем ниже листинге показано, как определить предостановочный обработчик HTTP GET в модуле.

Листинг 17.6. Фрагмент YAML предостановочного обработчика: pre-stop-hook-httpget.yaml

```
lifecycle:
  preStop:
    httpGet:
      port: 8080
      path: shutdown
```

Этот предостановочный обработчик выполняет запрос HTTP GET

Запрос отправляется в http://POD_IP:8080/shutdown

Предостановочный обработчик, определенный в этом списке, выполняет запрос HTTP GET в http://POD_IP:8080/shutdown, как только агент Kubelet начинает завершать работу контейнера. Помимо порта `port` и пути `path`, показанных в листинге, вы также можете задать поля `scheme` (HTTP или HTTPS) и `host`, а еще `httpHeaders`, которые должны быть отправлены в запросе. Поле `host` по умолчанию обозначает IP-адрес модуля. Только не назначайте ему значение localhost, потому что localhost будет ссылаться на узел, а не на модуль.

В отличие от постстартового обработчика, контейнер будет завершен независимо от результата этого обработчика – код отклика HTTP с ошибкой или ненулевой код выхода при использовании командного обработчика не будет препятствовать завершению работы контейнера. Если предостановочный об-

работчик не срабатывает, то среди событий модуля вы увидите предупреждение `FailedPreStopHook`, и поскольку модуль вскоре после этого удаляется (в конце концов, с самого начала удаление этого модуля как раз и вызвало предостановочный обработчик), вы можете даже не заметить, что предостановочному обработчику не удалось выполниться правильно.

> **СОВЕТ.** Если успешное завершение работы предостановочного обработчика имеет решающее значение для правильной работы системы, проверьте, выполняется ли он вообще. Я был свидетелем ситуаций, когда предостановочный обработчик не работал и разработчик даже не знал об этом.

Использование предостановочного обработчика, поскольку ваше приложение не получает сигнала SIGTERM

Многие разработчики делают ошибку, определяя предостановочный обработчик исключительно для отправки сигнала `SIGTERM` своим. Они делают это, потому что они не видят, что их приложение получает сигнал `SIGTERM`, который отправляется агентом Kubelet. Причина, по которой сигнал не принимается приложением, заключается не в том, что Kubernetes не отправляет его, а в том, что сигнал не передается процессу приложения внутри самого контейнера. Если образ контейнера настроен на запуск оболочки, которая, в свою очередь, запускает процесс приложения, то сигнал может быть съеден самой оболочкой, а не передан дочернему процессу.

В таких случаях, вместо того чтобы добавлять предостановочный обработчик для отправки сигнала непосредственно в приложение, правильное решение заключается в том, чтобы убедиться, что оболочка передает сигнал приложению. Это может быть достигнуто путем обработки сигнала в сценарии оболочки, запущенном в качестве главного процесса контейнера, а затем передачи его приложению. Либо вы можете вообще не сконфигурировать образ контейнера для запуска оболочки и вместо этого напрямую запускать двоичный файл приложения. Это можно сделать с помощью формы исполнения `ENTRYPOINT` или `CMD` в файле Dockerfile: `ENTRYPOINT ["/mybinary"]` вместо `ENTRYPOINT /mybinary`.

Контейнер с использованием первой формы запускает исполняемый файл `mybinary` в качестве своего главного процесса, в то время как вторая форма запускает оболочку в качестве главного процесса, а процесс `mybinary` выполняется как дочерний процесс оболочки.

Обработчики жизненного цикла нацелены на контейнеры, а не модули

В качестве заключительной мысли о постстартовых и предостановочных обработчиках следует подчеркнуть, что обработчики жизненного цикла относятся не к модулям, а к контейнерам. Не следует использовать предостановочный обработчик для выполнения действий, которые необходимо выполнить

при завершении модуля. Причина в том, что предостановочный обработчик вызывается, когда контейнер терминируется (скорее всего, из-за неудачной проверки живучести). Это может произойти несколько раз в жизни модуля, не только когда модуль находится в процессе выключения.

17.2.5 Выключение модуля

Мы затронули тему терминации модуля, поэтому давайте рассмотрим эту тему более подробно и обсудим, что именно происходит во время выключения модуля. Это важно для понимания того, каким образом чисто завершать работу работающего в модуле приложения.

Давайте начнем с самого начала. Завершение работы модуля инициируется удалением объекта Pod через сервер API. При получении запроса HTTP DELETE сервер API пока объект не удаляет, а только устанавливает в нем поле `deletionTimestamp`. Модули, в которых установлено поле `deletionTimestamp`, находятся в процессе терминирования.

Как только агент Kubelet замечает, что модуль должен быть терминирован, он начинает завершать работу каждого контейнера модуля. Он дает каждому контейнеру время, чтобы тот выключился корректно, но это время ограничено. Это время называется льготным периодом терминации и настраивается для каждого модуля. Таймер запускается, как только начинается процесс терминирования. Затем выполняется следующая последовательность событий.

1. Запустить предостановочный обработчик, если он сконфигурирован, и дождаться его завершения.
2. Отправить сигнал `SIGTERM` в главный процесс контейнера.
3. Подождать до тех пор, пока контейнер не выключится полностью или пока не закончится льготный период терминации.
4. Принудительно завершить процесс с помощью `SIGKILL`, если он еще не завершил работу корректно.

Данная последовательность событий показана на рис. 17.5.

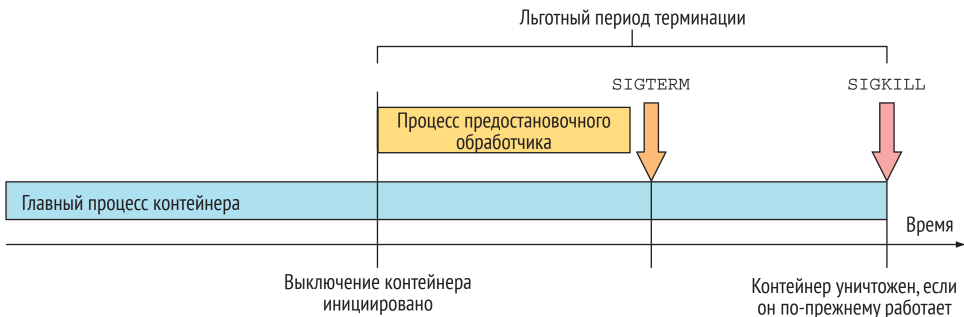


Рис. 17.5. Последовательность терминации контейнера

Указание льготного периода терминации

Льготный период терминации может быть сконфигурирован в спецификации модуля путем определения поля `spec.terminationGracePeriodSeconds`. По

умолчанию значение поля равно 30, то есть контейнерам модуля будет предоставлено 30 секунд для корректного завершения работы, прежде чем они будут уничтожены принудительно.

> **СОВЕТ.** Вы должны установить достаточно длительный льготный период, чтобы за это время ваш процесс мог завершить очистку.

Льготный период, указанный в спецификации модуля, также может быть переопределен при удалении модуля следующим образом:

```
$ kubectl delete po mypod --grace-period=5
```

Эта команда проинструктирует агента Kubelet ждать пять секунд, чтобы модуль выключился чисто. Когда все контейнеры модуля остановлены, Kubelet уведомляет об этом сервер API, и ресурс модуля будет окончательно удален. Вы можете заставить сервер API немедленно удалить ресурс, не дожидаясь подтверждения, установив льготный период равным нулю и добавив параметр `--force`, следующим образом:

```
$ kubectl delete po mypod --grace-period=0 --force
```

Будьте осторожны при использовании этого параметра, особенно с модулями StatefulSet. Контроллер набора модулей с внутренним состоянием делает все возможное, чтобы никогда не запускать два экземпляра того же модуля одновременно (два модуля с одинаковым порядковым индексом и именем и прикрепленные к одинаковому постоянному тому PersistentVolume). Принудительное удаление модуля приведет к тому, что контроллер создаст сменный модуль, не дожидаясь завершения работы контейнеров удаленного модуля. Другими словами, два экземпляра одного модуля могут работать одновременно, что может привести к некорректной работе кластера с внутренним состоянием. Удаляйте модули с внутренним состоянием принудительно, только когда вы абсолютно уверены, что модуль больше не работает или не может обмениваться с другими членами кластера (вы можете быть уверены в этом, когда у вас есть подтвержденные сведения о том, что узел, который разместил у себя модуль, отказал или был отключен от сети и не может повторно подключиться).

Теперь, когда вы понимаете, как контейнеры выключаются, давайте посмотрим на это с точки зрения приложения и рассмотрим, как приложения должны обрабатывать процедуру терминации.

Реализация надлежащего обработчика выключения в приложении

Приложения должны реагировать на сигнал `SIGTERM`, начиная процедуру выключения и завершая по ее окончании. Вместо того чтобы обрабатывать сигнал `SIGTERM`, приложение может быть уведомлено о выключении через предостановочный обработчик. В обоих случаях приложение затем имеет лишь фиксированное количество времени на чистую терминацию.

Но что делать, если вы не можете предсказать, сколько времени потребуется приложению, чтобы выключиться чисто? Например, представьте, что ваше приложение является распределенным хранилищем данных. При уменьшении масштаба один из экземпляров модуля будет удален и, следовательно, выключен. В процедуре выключения модуль должен перенести все свои данные в оставшиеся модули, чтобы обеспечить их сохранность. Должен ли модуль начать перенос данных после получения сигнала завершения (либо через сигнал SIGTERM, либо через предостановочный обработчик)?

Абсолютно нет! Это не рекомендуется, по крайней мере, по следующим двум причинам:

- терминация контейнера не обязательно означает, что весь модуль терминируется;
- вы не имеете никаких гарантий, что процедура выключения закончится прежде, чем процесс будет уничтожен.

Этот второй сценарий не происходит только тогда, когда льготный период закончится до того, как приложение закончит выключение корректно, но и тогда, когда выполняющий модуль узел аварийно завершает работу, находясь не в середине последовательности выключения контейнера. Даже если узел затем снова запустится, агент Kubelet не перезапускает процедуру выключения (он даже не запустит контейнер снова). Совершенно нет никакой гарантии, что модулю будет предоставлена возможность завершить всю свою процедуру выключения.

Замена критических процедур выключения специальными модулями с процедурой выключения

Как убедиться, что критическая процедура выключения, которая непременно должна выполняться до полного завершения, выполняется до полного завершения (например, чтобы обеспечить миграцию данных модуля в другие модули)?

Одно из решений заключается в том, чтобы приложение (после получения сигнала о терминации) создало новый ресурс задания Job, который запустит новый модуль, единственная задача которого заключается в миграции данных удаленного модуля в оставшиеся модули. Но если вы обратили внимание, то будете знать, что у вас нет никакой гарантии, что приложению действительно удастся создавать объект Job абсолютно каждый раз. Что делать, если узел аварийно прекращает работу ровно тогда, когда приложение пытается это сделать?

Правильный способ справиться с этой проблемой – иметь выделенный, постоянно работающий модуль, который непрерывно проверяет наличие подвисших данных. Когда этот модуль находит подвисшие данные, он может их перенести в остальные модули. Вместо постоянно работающего модуля можно также использовать ресурс CronJob и периодически запускать модуль.

Вы можете подумать, что объекты с внутренним состоянием StatefulSet могут здесь помочь, но это не так. Как вы помните, уменьшение масштаба на-

бора модулей с внутренним состоянием StatefulSet оставляет не у дел заявки PersistentVolumeClaim, бросая на произвол судьбы данные, хранящиеся в постоянном томе PersistentVolume. Разумеется, при последующем увеличении масштаба постоянный том будет повторно присоединен к новому экземпляру модуля, но что, если это увеличение масштаба никогда не произойдет (или произойдет через длительное время)? По этой причине вам может понадобиться запустить модуль миграции данных также при использовании наборов StatefulSet (этот сценарий показан на рис. 17.6). Для того чтобы предотвратить миграцию во время обновления приложения, модуль миграции данных может быть сконфигурирован на ожидание, чтобы предоставить модулю с внутренним состоянием время для повторного запуска перед выполнением миграции.

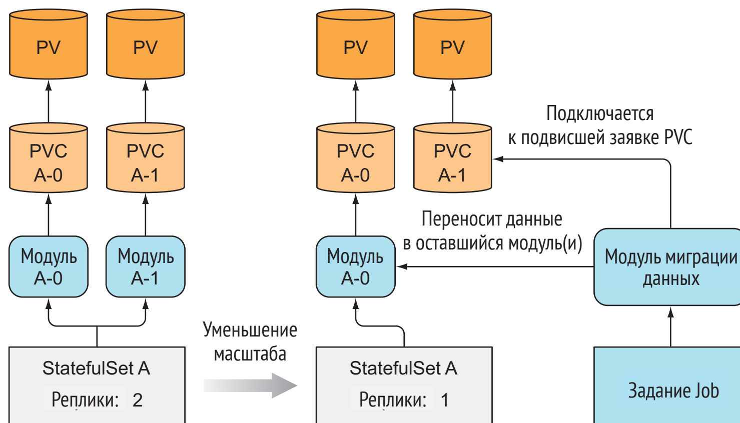


Рис. 17.6. Использование выделенного модуля для миграции данных

17.3 Обеспечение правильной обработки всех клиентских запросов

Теперь у вас есть хорошее понимание того, как делать так, чтобы модули выключались чисто. Теперь мы рассмотрим жизненный цикл модуля с точки зрения клиентов модуля (клиентов, потребляющих службы, которые модуль предоставляет). Это важно понимать, если вы не хотите, чтобы клиенты столкнулись с проблемами при увеличении или уменьшении масштаба модулей.

Само собой разумеется, что вы хотите, чтобы все запросы клиентов обрабатывались должным образом. Вы, очевидно, не хотите видеть прерванные подключения, когда модули запускаются или выключаются. Сама по себе система Kubernetes не предотвращает такое развитие событий. И ваше приложение должно следовать нескольким правилам, чтобы предотвращать прерывание подключений. Прежде всего давайте сосредоточимся на том, чтобы гарантировать, что все подключения обрабатываются правильно, когда модуль запускается.

17.3.1 Предотвращение прерывания клиентских подключений при запуске модуля

Если вы понимаете, как работают службы и конечные точки служб, то обеспечить правильную обработку каждого подключения при запуске модуля будет простым делом. При запуске модуль добавляется в качестве конечной точки во все службы, селектор меток которых соответствует меткам модуля. Как вы, возможно, помните из главы 5, модуль также должен подать сигнал системе Kubernetes, что он готов. До тех пор, пока это не произойдет, он не станет конечной точкой службы и поэтому не будет получать запросы от клиентов.

Если в спецификации модуля не указана проверка готовности, то модуль всегда считается готовым. Он начнет получать запросы почти сразу – как только первый kube-proxy обновит правила `iptables` на своем узле и первый клиентский модуль попытается подключиться к службе. Если к тому времени ваше приложение не будет готово принимать подключения, то клиенты увидят ошибки «отказано в подключении».

В этой ситуации вам нужно лишь обеспечить, чтобы ваша проверка готовности возвращала успех только тогда, когда ваше приложение готово правильно обрабатывать входящие запросы. Хороший первый шаг – добавить модуль готовности HTTP GET и указать его на базовый URL-адрес приложения. Во многих случаях этого будет вполне достаточно, чтобы избавить от необходимости реализовывать в приложении специальную конечную точку готовности.

17.3.2 Предотвращение прерванных подключений при выключении модуля

Теперь давайте посмотрим, что происходит на другом конце жизни модуля – когда модуль удаляется и его контейнеры терминируются. Мы уже отмечали то, как контейнеры модуля должны начинать чистое выключение, как только они получают сигнал `SIGTERM` (или когда будет выполнен ее предостановочный обработчик). Но гарантирует ли это, что все запросы клиентов обрабатываются должным образом?

Как должно вести себя приложение, когда оно получает сигнал терминации? Следует ли ему продолжать принимать запросы? Как насчет запросов, которые уже получены, но еще не завершены? Как насчет постоянных HTTP-подключений, которые могут находиться между запросами, но являются открытыми (когда в подключении не существует активного запроса)? Прежде чем мы сможем ответить на эти вопросы, нам нужно подробно рассмотреть цепочку событий, которая разворачивается в кластере при удалении модуля.

Последовательность событий, происходящих при удалении модуля

В главе 11 мы тщательно рассмотрели, какие компоненты составляют кластер Kubernetes. Необходимо всегда помнить, что эти компоненты выполняются как отдельные процессы на множестве машин. Они не являются частью единого большого монолитного процесса. Требуется время, чтобы все ком-

поненты находились на одной стороне относительно состояния кластера. Давайте рассмотрим этот факт, посмотрев, что происходит в кластере при удалении модуля.

Когда запрос на удаление модуля получен сервером API, он сначала изменяет состояние в хранилище etcd, а затем уведомляет своих наблюдателей об удалении. Среди этих наблюдателей – агент Kubelet и контроллер конечных точек. На рис. 17.7 показаны две последовательности событий (помеченные как A или B), происходящих параллельно.

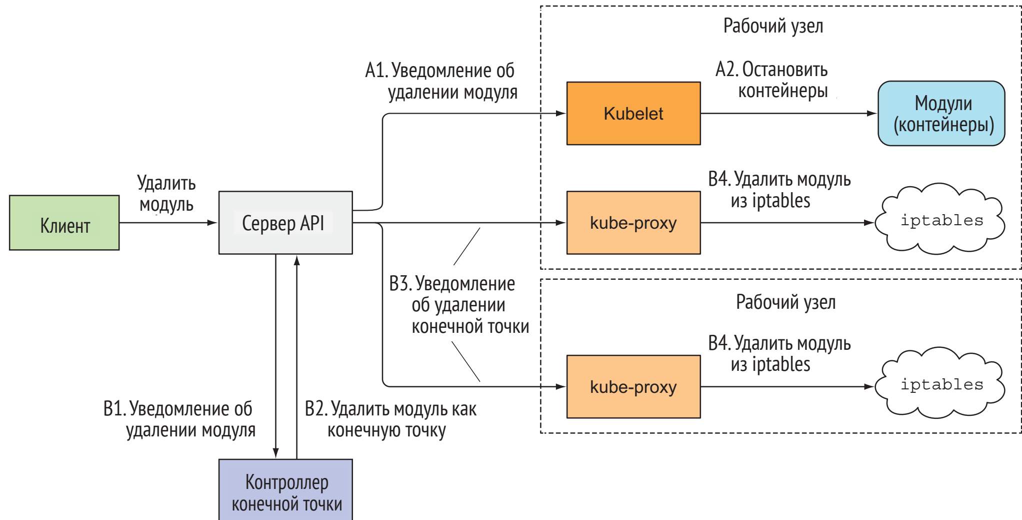


Рис. 17.7. Последовательность событий, происходящих при удалении модуля

В последовательности событий A вы видите, что как только Kubelet получает уведомление о том, что модуль должен быть терминирован, он инициирует последовательность его выключения, как описано в разделе 17.2.5 (запустить предостановочый обработчик, отправить сигнал `SIGTERM`, подождать некоторое время, а затем принудительно уничтожить контейнер, если он еще не выключился самостоятельно). Если приложение откликается на сигнал `SIGTERM`, немедленно прекращая получать клиентские запросы, то любой клиент, пытающийся к нему подключиться, получит сообщение об ошибке отказа в подключении. Необходимое для этого время с момента удаления модуля – относительно короткое из-за прямого пути от сервера API к агенту Kubelet.

Теперь давайте посмотрим, что происходит в другой последовательности событий – той, которая ведет к тому, что модуль удаляется из правил `iptables` (последовательность B на рисунке). Когда контроллер конечных точек (который выполняется в диспетчере контроллера в плоскости управления Kubernetes) получает уведомление об удаляемом модуле, он удаляет модуль как конечную точку во всех службах, частью которых модуль является. Это делается изменением объекта API конечных точек Endpoints путем отправки запроса REST на сервер API. Затем сервер API уведомляет всех клиентов, наблюдающих за объектом Endpoints. Среди этих наблюдателей все прокси kube-proxy, рабо-

тающие на рабочих узлах. Каждый из этих прокси затем обновляет правила `iptables` на своем узле, что предотвращает перенаправление новых подключений в терминируемый модуль. Важная деталь здесь состоит в том, что удаление правил `iptables` не влияет на существующие подключения – клиенты, которые уже подключены к модулю, все равно будут отправлять дополнительные запросы к модулю через эти существующие подключения.

Обе эти последовательности событий происходят параллельно. Скорее всего, время, необходимое для выключения процесса приложения в модуле, немного меньше времени, необходимого для обновления правил `iptables`. Цепочка событий, которая приводит к обновлению правил `iptables`, значительно длиннее (см. рис. 17.8), потому что событие должно сначала достигнуть контроллера конечных точек, который затем отправляет новый запрос на сервер API, и потом сервер API должен уведомить kube-proxy, прежде чем этот прокси наконец изменит правила `iptables`. Существует высокая вероятность того, что сигнал `SIGTERM` будет отправлен задолго до обновления правил `iptables` на всех узлах.

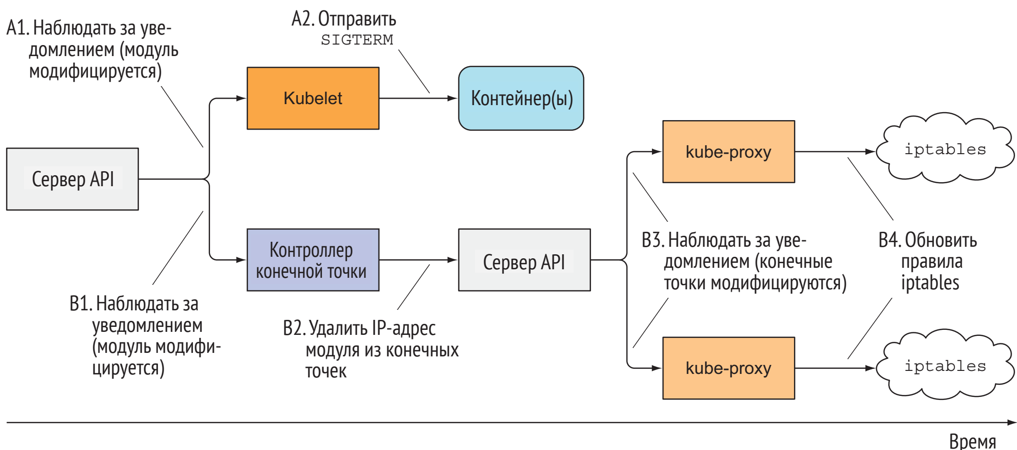


Рис. 17.8. Временная шкала событий при удалении модуля

В конечном счете модуль может все еще получать клиентские запросы, после того как ему был отправлен сигнал о терминации. Если приложение закрывает серверный сокет и немедленно прекращает принимать подключения, то это приведет к тому, что клиенты получат ошибки типа «отказано в подключении» (аналогично тому, что происходит при запуске модуля, если ваше приложение не способно принимать подключения немедленно и вы не определяете для него проверку готовности).

Решение проблемы

Поиск решений этой проблемы создает впечатление, что если в ваш модуль добавить проверку готовности, то проблема будет решена. Как предполагается, вам нужно лишь сделать так, чтобы проверка готовности начала не сраба-

тывать, как только модуль получает сигнал SIGTERM. Это предположительно приведет к тому, что модуль будет удален в качестве конечной точки службы. Однако удаление произойдет только после того, как проверка готовности не сработает несколько раз подряд (это конфигурируется в спецификации проверки готовности). И тогда, очевидным образом, удаление по-прежнему нуждается в том, чтобы достигнуть kube-proxy до того, как модуль будет удален из правил iptables.

На самом деле проверка готовности не имеет абсолютно никакого отношения ко всему этому процессу. Контроллер конечных точек удаляет модуль из конечных точек службы, как только он получает уведомление об удалении модуля (когда поле deletionTimestamp в спецификации модуля больше не имеет значения null). С этого момента результат проверки готовности не имеет значения.

Каким же будет решение проблемы? Каким образом гарантированно обработать все запросы?

Ясно, что модулю требуется продолжать принимать подключения даже после того, как он получит сигнал о терминации, до тех пор, пока все kube-proxy не закончат обновление правил iptables. И дело не только в kube-proxy. Кроме него, также могут быть контроллеры Ingress или балансировщики нагрузки, перенаправляющие подключения непосредственно в модуль, не проходя через службу (iptables). Сюда также входят клиенты, использующие балансировку нагрузки на стороне клиента. Для того чтобы никто из клиентов не столкнулся с прерыванием подключения, вам придется подождать до тех пор, пока все прокси каким-либо образом вас не уведомят о том, что они больше не перенаправляют подключения в модуль.

Это невозможно, потому что все эти компоненты распределены по многим разным машинам. Даже если бы вы знали местоположение каждой из них и могли подождать до тех пор, пока все они сообщат, что не возражают против закрытия модуля, что вы будете делать, если один из них не будет откликаться? Как долго вы будете ждать отклика? Напомним, что в течение этого времени вы задерживаете процесс выключения.

Единственная разумная вещь, которую вы можете сделать, – это подождать достаточно долго, чтобы убедиться, что все прокси сделали свою работу. Но какой срок ожидания будет достаточным? В большинстве ситуаций нескольких секунд должно быть достаточно, но нет никакой гарантии, что этого будет достаточно каждый раз. Когда сервер API или контроллер конечных точек перегружен, то, для того чтобы уведомление достигло kube-proxy, может уйти больше времени. Важно понимать, что невозможно решить проблему идеально, но даже добавление 5- или 10-секундной задержки должно значительно улучшить опыт пользователя. Вы можете применить более длительную задержку, но не стоит перебарщивать, потому что задержка не даст выключить контейнер быстро и приведет к тому, что модуль будет отображаться в списках спустя длительное время, после того как он был удален, что всегда расстраивает пользователя, удаляющего модуль.

Завершение этого раздела

Резюмируя данную тему, правильное завершение работы приложения включает в себя следующие шаги:

- подождать несколько секунд, затем прекратить прием новых подключений;
- закрыть все постоянные (keep-alive) подключения, не находящиеся в середине запроса;
- дождаться завершения всех активных запросов;
- затем полностью закрыть.

Для того чтобы разобраться в том, что происходит с подключениями и запросами во время этого процесса, внимательно изучите рис. 17.9.

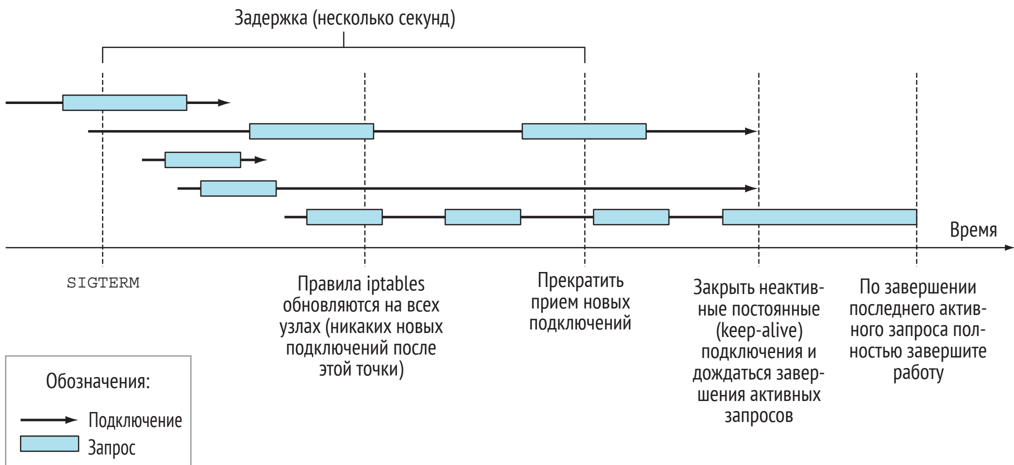


Рис. 17.9. Правильная обработка существующих и новых подключений после получения сигнала о терминации

Не так просто, как выйти из процесса сразу после получения сигнала о терминации, верно? Стоит ли проходить через весь этот процесс? Решать вам. Но самое меньшее, что вы можете сделать, – это добавить предостановочный обработчик, который ожидает несколько секунд, к примеру как в следующем ниже листинге.

Листинг 17.7. Предостановочный обработчик для предотвращения прерывания подключений

```
lifecycle:
  preStop:
    exec:
      command:
        - sh
        - -c
        - "sleep 5"
```

Благодаря этому вам вообще не нужно модифицировать программный код вашего приложения. Если ваше приложение уже гарантирует, что все находящиеся в процессе запросы будут обработаны полностью, это может быть все, что вам нужно.

17.4 Упрощение запуска приложений и управления ими в Kubernetes

Надеюсь, что теперь вы имеете более четкое представление о том, как делать так, чтобы ваши приложения корректно обрабатывали запросы клиентов. Теперь мы рассмотрим другие аспекты, касающиеся того, как собирать приложение так, чтобы им было легко управлять в Kubernetes.

17.4.1 Создание управляемых образов контейнеров

Когда вы упаковываете свое приложение в образ, вы можете решить включить исполняемый файл приложения и любые дополнительные библиотеки, которые ему нужны, либо вы можете упаковать всю файловую систему ОС вместе с приложением. Так поступают очень многие люди, хотя это обычно излишне.

Нужен ли вам каждый отдельный файл из дистрибутива ОС в образе? Наверное, нет. Большинство файлов никогда не будет использоваться и сделает образ крупнее, чем он должен быть. Безусловно, слоеная организация образов гарантирует, что каждый отдельный слой загружается только один раз, но нежелательна даже необходимость ждать дольше, чем нужно, в первый раз, когда модуль назначается узлу.

Развертывание новых модулей и их масштабирование должно быть быстрым. Это требует наличия небольших образов без ненужного хлама. Если вы создаете приложения с использованием языка Go, то образы не должны содержать ничего, кроме одного двоичного исполняемого файла приложения. Это делает Go-ориентированные образы контейнеров весьма небольшими и идеальными для Kubernetes.

> **СОВЕТ.** Для таких образов используйте директиву `FROM scratch` в файле Dockerfile.

Но на практике вы скоро увидите, что эти минимальные образы чрезвычайно трудно отлаживать. Когда вам нужно в первый раз запустить инструмент, такой как `ping`, `dig`, `curl`, или что-то подобное внутри контейнера, вы поймете, насколько важно, чтобы образы контейнеров также включали, по крайней мере, ограниченный набор этих инструментов. Я не могу сказать, что включать и что не включать в ваши образы, потому что это зависит от того, как вы все делаете, поэтому вам нужно будет найти золотую середину самостоятельно.

17.4.2 Правильное тегирование образов и рациональное использование политики imagePullPolicy

Вы также скоро узнаете, что обращение к тегу `latest` образа в манифестах модуля вызовет проблемы, потому что вы не можете сказать, какая версия образа выполняется каждой отдельной репликой модуля. Несмотря на то что изначально все реплики модуля выполняют одинаковую версию образа, если вы закачиваете новую версию образа под тегом `latest`, а затем модули переназначаются (либо вы увеличиваете масштаб своего развертывания), то новые модули будут работать в новой версии, в то время как старые будут работать в старой. Кроме того, использование тега `latest` делает невозможным откат к предыдущей версии (если вы снова не закачаете старую версию образа).

Использовать теги, содержащие правильное обозначение версии, вместо тега `latest`, за исключением, возможно, этапа разработки, почти обязательно. Имейте в виду, что если вы используете мутирующие теги (вы отправляете в реестр изменения на один и тот же тег), вам нужно будет установить поле `imagePullPolicy` в спецификации модуля, присвоив ему значение `Always`. Но если вы используете его в боевых средах, учитывайте, что с ним связан один большой нюанс. Если для политики выгрузки образа задано значение `Always`, то среда выполнения контейнера будет связываться с хранилищем образов при каждом развертывании нового модуля. Это немного замедляет запуск модуля, потому что узел должен проверять, был изменен образ или нет. Хуже того, эта политика не дает запустить модуль, когда не удается связаться с хранилищем.

17.4.3 Использование многомерных меток вместо одномерных

Не забудьте пометить все ваши ресурсы, а не только модули. Убедитесь, что к каждому ресурсу добавлено несколько меток, чтобы их можно было выбрать в каждой отдельной размерности. Вы (или системные администраторы) будете благодарны, что сделали это, когда количество ресурсов увеличится. Метки могут включать в себя такие вещи, как:

- имя приложения (или, возможно, микросервиса), которому ресурс принадлежит;
- слой приложения (фронтенд, бэкенд и т. д.);
- среда (разработка, контроль качества, промежуточный этап, рабочая среда и т. д.);
- версия;
- тип релиза (стабильный, канареечный, зеленый или синий для зеленых/синих развертываний и т. д.);
- клиент (при использовании отдельных модулей для каждого клиента вместо пространств имен);
- раздел (shard) для разделенных систем.

Это позволит вам управлять ресурсами в группах, а не по отдельности, и позволит легко увидеть, чему каждый ресурс принадлежит.

17.4.4 Описание каждого ресурса с помощью аннотаций

Для добавления дополнительных сведений в свои ресурсы используйте аннотации. Как минимум, ресурсы должны содержать аннотацию с описанием ресурса и аннотацию с контактной информацией ответственного за него лица.

В архитектуре микросервисов модули могут содержать аннотацию, в которой перечислены имена других используемых модулем служб. Это позволяет отображать зависимости между модулями. Другие аннотации могут включать сведения о сборке и версии, а также метаданные, используемые инструментами или графическими интерфейсами пользователя (имена значков и т. д.).

Как метки, так и аннотации намного упрощают управление работающими приложениями, но нет ничего хуже, когда приложение начинает сбоить, и вы не знаете, почему.

17.4.5 Предоставление информации о причинах прекращения процесса

Нет ничего более неприятного, чем выяснять, почему контейнер терминируется (или даже делает это непрерывно), в особенности если это происходит в самый неподходящий момент. Будьте добры к системным администраторам и облегчите их жизнь, включив в ваши журнальные файлы всю необходимую отладочную информацию.

Однако, для того чтобы сделать расследование еще проще, вы можете воспользоваться еще одним функциональным средством системы Kubernetes, которое позволяет показывать причину в статусе модуля, по которой контейнер прекратил работу. Для этого процесс должен записать сообщение о терминации в определенный файл в файловой системе контейнера. Содержимое этого файла считывается агентом Kubelet, когда контейнер завершает работу, и выводится в результатах команды `kubectl describe pod`. Если приложение использует этот механизм, то системный администратор может быстро понять, почему приложение завершило работу, даже не глядя на журналы контейнера.

Файлом, в который процесс должен по умолчанию записать сообщение, является /dev/termination-log, но его можно поменять, задав значение в поле `terminationMessagePath` в определении контейнера в секции `spec` модуля.

Это можно увидеть в действии, запустив модуль, контейнер которого немедленно умирает, как показано в следующем ниже листинге.

Листинг 17.8. Модуль, пишущий сообщение о терминации: termination-message.yaml

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

```
metadata:
  name: pod-with-termination-message
spec:
  containers:
  - image: busybox
    name: main
    terminationMessagePath: /var/termination-reason
    command:
    - sh
    - -c
    - 'echo "I''ve had enough" > /var/termination-reason ; exit 1'
```

Для файла сообщения о терминации вы переопределяете заданный по умолчанию путь

Контейнер запишет сообщение в файл непосредственно перед выходом

При запуске этого модуля вы вскоре увидите, что статус модуля будет показан как `CrashLoopBackOff`. Если вы затем выполните команду `kubectl describe`, то сможете увидеть, почему контейнер умер, без необходимости копаться в его журналах, как показано в следующем ниже листинге.

Листинг 17.9. Просмотр сообщения о терминации контейнера с помощью команды kubectl describe

```
$ kubectl describe po
Name: pod-with-termination-message
...
Containers:
...
    State:      Waiting
      Reason:   CrashLoopBackOff
    Last State: Terminated
      Reason:   Error
      Message:  I've had enough
      Exit Code:          1
      Started:            Tue, 21 Feb 2017 21:38:31 +0100
      Finished:           Tue, 21 Feb 2017 21:38:31 +0100
      Ready:              False
      Restart Count:      6
```

Вы можете увидеть причину, по которой контейнер умер, не проверяя его журналы

Как видите, сообщение «I’ve had enough» (С меня хватит), которое процесс написал в файл /var/termination-reason, показано в секции `Last State` контейнера. Обратите внимание, что этот механизм не ограничивается только аварийно прекращающими работу контейнерами. Его также можно использовать в модулях, которые запускают завершенную задачу и прекращают ее работу успешно (вы найдете соответствующий пример в файле termination-message-success.yaml).

Этот механизм отлично подходит для терминируемых контейнеров, но вы, вероятно, согласитесь, что аналогичный механизм также будет полезен для вывода специфичных для приложений сообщений о статусе не только терминируемых, но и работающих контейнеров. В настоящее время Kubernetes не

предоставляет такой функциональности, и я не знаю о каких-либо планах по ее внедрению.

> **ПРИМЕЧАНИЕ.** Если контейнер не может написать сообщение в любой файл, вы можете задать значение `FallbackToLogsOnError` в поле `terminationMessagePolicy`. В этом случае последние несколько строк журнала событий контейнера используются в качестве сообщения о терминации (но только тогда, когда контейнер завершает свою работу неуспешно).

17.4.6 Работа с журналами приложений

Раз уж мы говорим о теме ведения журнала событий приложения, давайте повторим, что приложения должны писать в стандартный вывод, а не в файлы. Это упрощает просмотр журналов с помощью команды `kubectl logs`.

> **СОВЕТ.** Если контейнер аварийно завершает работу и заменяется новым, то вы увидите журнал нового контейнера. Для просмотра журналов предыдущего контейнера используйте параметр `--previous` вместе с командой `kubectl logs`.

Если приложение выполняет вывод журнальных данных в файл, а не в стандартный вывод, вы можете вывести журнальный файл с помощью альтернативного подхода:

```
$ kubectl exec <модуль> cat <файл-журнала>
```

Здесь инструмент `kubectl` исполняет команду `cat` внутри контейнера, передающую журналы обратно в `kubectl`, который печатает их в вашем терминале.

Копирование журнала и других файлов в контейнер и из него

Вы также можете скопировать журнальный файл на свою локальную машину с помощью команды `kubectl cp`, которую мы еще не рассматривали. Она позволяет копировать файлы из контейнера и в контейнер. Например, если имеется модуль с именем `foo-pod` и его единственный контейнер содержат файл в `/var/log/foo.log`, то вы можете перенести его на локальную машину с помощью следующей ниже команды:

```
$ kubectl cp foo-pod:/var/log/foo.log foo.log
```

Для того чтобы скопировать файл с локальной машины в модуль, укажите имя модуля во втором аргументе:

```
$ kubectl cp localfile foo-pod:/etc/remotefile
```

Эта команда копирует файл `localfile` в удаленный файл /etc/remotefile внутри контейнера модуля. Если модуль содержит несколько контейнеров, укажите контейнер с помощью параметра `-c containerName`.

Использование централизованного журналирования

В боевой системе вам, как правило, потребуется использовать централизованное решение для ведения журналов на уровне кластера, для того чтобы все журналы собирались и (постоянно) хранились в центральном расположении. Это позволяет просматривать исторические журналы и анализировать тенденции. Без такой системы журналы модуля доступны только тогда, когда модуль существует. Как только он удаляется, его журналы тоже удаляются.

Система Kubernetes как таковая не обеспечивает какого-либо централизованного журналирования. Компоненты, необходимые для обеспечения централизованного хранения и анализа всех журналов контейнеров, должны предоставляться дополнительными компонентами, которые обычно выполняются как обычные модули в кластере.

Разворачивать решения централизованного журналирования достаточно легко. Вам нужно лишь развернуть несколько манифестов YAML/JSON, и все готово. На движке Google Kubernetes Engine это еще проще. При настройке кластера установите флажок Enable Stackdriver Logging (Включить журналирование Stackdriver). Настройка централизованного журналирования на локальном кластере Kubernetes выходит за рамки этой книги, но я дам вам краткий обзор того, как это обычно делается.

Возможно, вы уже слышали о стеке ELK, состоящем из хранилища ElasticSearch, серверного конвейера обработки данных Logstash и веб-инструмента Kibana. Слегка измененным его вариантом является стек EFK, где конвейер Logstash заменен на FluentD.

При использовании стека EFK для централизованного журналирования каждый узел кластера Kubernetes запускает агента FluentD (как правило, в качестве модуля, развернутого посредством объекта DaemonSet), который отвечает за сбор журналов событий из контейнеров, тегируя их модульно-специфической информацией и доставляя их в хранилище ElasticSearch, которое сохраняет их постоянно. ElasticSearch тоже развертывается как модуль где-то в кластере. Журналы можно просматривать и анализировать в веб-браузере через Kibana, веб-инструмент для визуализации данных хранилища ElasticSearch. Он также обычно работает как модуль, и доступ к нему предоставляется через службу. Три компонента стека EFK показаны на рис. 17.10.

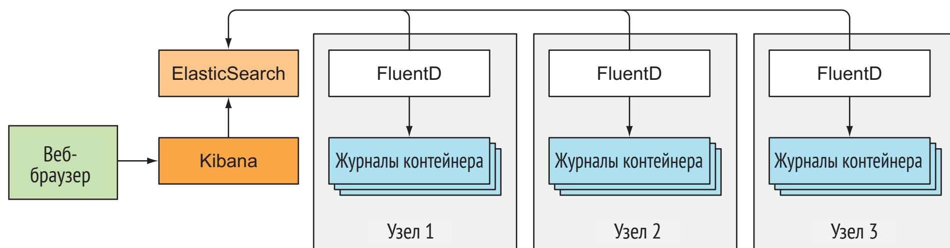


Рис. 17.10. Централизованное журналирование с помощью FluentD, ElasticSearch и Kibana

ПРИМЕЧАНИЕ. В следующей главе вы познакомитесь со схемами менеджера пакетов Helm. Схемы, созданные сообществом Kubernetes, можно использовать для развертывания стека EFK вместо создания собственных манифестов YAML.

Обработка многострочных инструкций журналирования

Агент FluentD хранит каждую строку файла журнала как запись в хранилище данных ElasticSearch. С этим есть одна проблема. Инструкции журналирования, охватывающие несколько строк, такие как трассировки стека исключений в Java, отображаются в централизованной системе журналирования как отдельные записи.

Для того чтобы решить эту проблему, вы можете сделать так, чтобы приложения выводили JSON вместо обычного текста. Благодаря этому многострочная инструкция журналирования может быть сохранена и показана в Kibana как одна запись. Но это делает просмотр журналов с помощью команды `kubectl logs` гораздо менее понятным для людей.

Решением может быть вывод удобочитаемых журналов в стандартный вывод, при этом записывая журналы JSON в файл с их обработкой в FluentD. Для этого необходимо соответствующим образом настроить агент FluentD на уровне узла или добавить в каждый модуль побочный контейнер журналирования.

17.5 Рекомендации по разработке и тестированию

Мы уже говорили о том, что нужно помнить при разработке приложений, но мы не говорили о разработке и тестировании рабочих процессов, которые помогут вам оптимизировать эти процессы. Не буду вдаваться в подробности, потому что каждый должен найти то, что лучше всего подходит для него. Перечислю лишь несколько отправных точек.

17.5.1 Запуск приложений за пределами Kubernetes во время разработки

Когда вы разрабатываете приложение, которое будет работать в производственном кластере Kubernetes, означает ли это, что вам также нужно запускать его в Kubernetes во время разработки? Не совсем. Необходимость создавать приложение после каждого незначительного изменения, затем создавать образ контейнера, отправлять его в реестр, а потом повторно развертывать модули – все это сделает разработку медленной и болезненной. К счастью, вам не нужно проходить через все эти трудности.

Вы всегда можете разрабатывать и запускать приложения на локальной машине, как вы привыкли. В конце концов, приложение, запущенное в Kubernetes, является обычным (хотя и изолированным) процессом, запущенным на одном из узлов кластера. Если приложение зависит от определенных

функциональных средств, предоставляемых средой Kubernetes, то вы можете легко реплицировать эту среду на машине для разработки.

Я даже не говорю о запуске приложения в контейнере. В большинстве случаев это не требуется – обычно приложение можно запускать непосредственно из среды IDE.

Подключение к внутренним службам

Если приложение в производственной среде подключается к бэкенд-службе и использует переменные среды `BACKEND_SERVICE_HOST` и `BACKEND_SERVICE_PORT` для поиска координат службы, очевидно, что эти переменные среды можно задать на локальной машине вручную и указать их на внутреннюю службу, независимо от того, выполняется она вне или внутри кластера Kubernetes. Если он выполняется внутри Kubernetes, то вы всегда можете (по крайней мере, временно) сделать службу доступной извне, поменяв ее на службу типа `NodePort` или `LoadBalancer`.

Подключение к серверу API

Аналогичным образом, если приложению требуется доступ к серверу API Kubernetes при работе в кластере Kubernetes, оно может легко взаимодействовать с сервером API извне кластера во время разработки. Если для самоаутентификации используется токен учетной записи ServiceAccount, то вы всегда можете скопировать файлы секрета учетной записи ServiceAccount на локальную машину с помощью команды `kubectl cp`. Серверу API все равно, находится обращающийся к нему клиент внутри кластера или за его пределами.

Если приложение использует контейнер-посредник, как описано в главе 8, то вам даже не нужны эти файлы секрета. Запустите прокси `kubectl-proxy` на локальной машине, запустите приложение локально, и оно должно быть готово к взаимодействию с вашим локальным прокси `kubectl-proxy` (в случае если оно и контейнер-посол привязывают прокси к одинаковому порту).

В этом случае необходимо убедиться, что учетная запись пользователя, применяемая локальным `kubectl`, имеет те же права, что и учетная запись ServiceAccount, под которой будет работать приложение.

Запуск внутри контейнера даже во время разработки

Когда во время разработки вам по какой-либо причине крайне необходимо запустить приложение в контейнере, есть способ избежать необходимости каждый раз создавать образ контейнера. Вместо того чтобы впечатывать двоичные файлы в образ, вы всегда можете подключить локальную файловую систему к контейнеру, например через тома платформы Docker. Благодаря этому, после того как вы соберете новую версию двоичных файлов приложения, от вас требуется только запустить контейнер заново (или даже это не придется делать, если поддерживается горячее переразвертывание). Нет никакой необходимости собирать образ заново.

17.5.2 Использование Minikube в разработке

Как вы можете видеть, ничто не заставляет вас запускать приложение внутри Kubernetes во время разработки. Но вы можете сделать это в любом случае, чтобы увидеть, как приложение ведет себя в реальных условиях Kubernetes.

Возможно, что для запуска примеров в этой книге вы использовали Minikube. Хотя кластер Minikube работает только на одном рабочем узле, тем не менее это ценный метод опробирования приложения в Kubernetes (и, конечно же, разработки всех манифестов ресурсов, составляющих полное приложение). Minikube не предлагает все, что обычно предоставляет настоящий многоузловой кластер Kubernetes, но в большинстве случаев это не имеет значения.

Монтирование локальных файлов в виртуальную машину Minikube и затем в контейнеры

Когда вы ведете разработку в Minikube и хотели бы испытывать каждое изменение в своем приложении в кластере Kubernetes, вы можете смонтировать вашу локальную файловую систему в виртуальную машину Minikube с помощью команды `minikube mount`, а затем смонтировать ее в свои контейнеры через том `hostPath`. Дополнительные инструкции о том, как это сделать, вы найдете в документации Minikube по адресу https://github.com/kubernetes/minikube/tree/master/docs.

Использование демона Docker внутри виртуальной машины Minikube для сборки образов

Если вы разрабатываете свое приложение в Minikube и планируете собирать образ контейнера после каждого изменения, то для выполнения сборки вы можете использовать демона Docker внутри виртуальной машины Minikube, вместо того чтобы собирать образ посредством локального демона Docker, отправки его в реестр, а затем его изъятия из реестра демоном в виртуальной машине. Для того чтобы использовать демон Docker инструмента Minikube, от вас требуется лишь указать на него переменную среды `DOCKER_HOST`. К счастью, это намного проще, чем кажется. Все, что вам нужно сделать, – это выполнить следующую ниже команду на локальной машине:

```
$ eval $(minikube docker-env)
```

Эта команда установит за вас все необходимые переменные среды. Затем вы собираете образы таким же образом, как если бы демон Docker работал на вашей локальной машине. После сборки образа его не нужно никуда загружать, так как он уже хранится локально на виртуальной машине Minikube, что означает, что новые модули могут сразу использовать этот образ. Если модули уже запущены, то, для того чтобы они были перезапущены, необходимо либо их удалить, либо уничтожить их контейнеры.

Локальная сборка образов и их копирование на виртуальную машину Minikube напрямую

Если вы не можете для сборки образов использовать демон внутри виртуальной машины, то у вас по-прежнему есть способ избежать принудительной загрузки образа в хранилище и запуска агента Kubelet на виртуальной машине Minikube. Если образ создается на локальной машине, то его можно скопировать на виртуальную машину Minikube с помощью следующей ниже команды:

```
$ docker save <образ> | (eval $(minikube docker-env) && docker load)
```

Как и прежде, образ сразу готов к использованию в модуле. Но убедитесь, что значение параметра `imagePullPolicy` в спецификации модуля не равно `Always`, так как это приведет к повторной выгрузке образа из внешнего реестра и потере скопированных изменений.

Сочетание Minikube с обычным кластером Kubernetes

У вас практически нет ограничений при разработке приложений с помощью Minikube. Вы даже можете объединить кластер Minikube с обычным кластером Kubernetes. Иногда я выполняю процессы в версии для разработки в локальном кластере Minikube и организовываю работу так, чтобы они обменивались с процессами, развернутыми в удаленном многоузловом кластере Kubernetes, расположенном на удалении в тысячи километров.

После того как я заканчиваю разработку, я могу переместить свои локальные наработки в удаленный кластер без каких-либо изменений и совершенно беспроблемно, благодаря тому что Kubernetes абстрагирует базовую инфраструктуру от приложения.

17.5.3 Версионирование и автоматическое развертывание ресурсных манифестов

Поскольку Kubernetes использует декларативную модель, вам никогда не придется выяснять текущее состояние развернутых ресурсов и выполнять императивные команды, для того чтобы привести это состояние к желаемому. От вас требуется только сообщить Kubernetes желаемое состояние, и он будет принимать все необходимые меры, для того чтобы согласовать состояние кластера с желаемым состоянием.

Коллекция ресурсных манифестов может храниться в системе управления версиями, что позволяет выполнять проверку кода, вести журнал аудита и при необходимости откатывать изменения. После каждой фиксации изменений можно выполнить команду `kubectl apply`, с тем чтобы изменения были отражены в развернутых ресурсах.

Если вы запускаете агента, который периодически (или при обнаружении новой фиксации) выгружает манифесты из системы управления версиями (Version Control System, VCS), а затем выполняет команду `apply`, то вы можете управлять работающими приложениями, просто фиксируя изменения в VCS

без необходимости вручную обращаться к серверу API Kubernetes. К счастью, люди в Box (который по случайности был использован для размещения рукописи этой книги и других материалов) разработали и выпустили инструмент под названием `kube-applier`, который делает именно то, что я описал. Исходный код инструмента находится по адресу https://github.com/box/kube-applier.

Для развертывания манифестов в кластере разработки, контроля качества в промежуточном и рабочем кластерах вы можете использовать несколько ветвей (или в разных пространствах имен в одном и том же кластере).

17.5.4 Знакомство с Ksonnet как альтернативой написанию манифестов YAML/JSON

В всей книге мы встретили целый ряд манифестов YAML. Я не считаю написание YAML слишком большой проблемой, в особенности когда вы узнаете, как использовать инструмент `kubectl explain`, чтобы увидеть доступные варианты. Однако некоторые люди считают это проблемой.

Как раз когда я заканчивал работу над рукописью этой книги, был анонсирован новый инструмент под названием Ksonnet. Он представляет собой библиотеку, надстроенную поверх языка шаблонизации данных Jsonnet, предназначенного для построения структур данных JSON. Вместо того чтобы писать полный JSON вручную, он позволяет определять параметризованные фрагменты JSON, давать им имя, а затем собирать полный манифест JSON, ссылаясь на эти фрагменты по имени, а не повторять один и тот же код JSON в нескольких местах – так же, как вы используете функции или методы в языке программирования.

Библиотека Ksonnet определяет фрагменты, которые можно найти в манифестах ресурсов Kubernetes, позволяя вам быстро собирать полный манифест JSON ресурсов Kubernetes с гораздо меньшим количеством кода. Соответствующий пример приведен в следующем ниже листинге.

Листинг 17.10. Развертывание kubia, написанное с помощью Ksonnet: kubia.ksonnet

```
local k = import "../ksonnet-lib/ksonnet.beta.1/k.libsonnet";

local container = k.core.v1.container;
local deployment = k.apps.v1beta1.deployment;

local kubiaContainer =
  container.default("kubia", "luksa/kubia:v1") +
  container.helpers.namedPort("http", 8080);

deployment.default("kubia", kubiaContainer) +
deployment.mixin.spec.replicas(3)
```

Это определяет контейнер под названием kubia, который использует образ luksa/kubia:v1 и включает порт под названием http

Это будет расширено в полный ресурс развертывания Deployment. Определенный здесь контейнер kubia будет включен в шаблон модуля развертывания

Показанный в листинге файл kubia.ksonnet конвертируется в полный манифест JSON развертывания Deployment при выполнении следующей ниже команды:

```
$ jsonnet kubia.ksonnet
```

Мощь библиотек Ksonnet и Jsonnet становится очевидной, когда вы понимаете, что можете определять свои собственные фрагменты более высокого уровня и делать все ваши манифесты согласованными и без дублирования. Более подробную информацию об использовании и установке библиотек Ksonnet и Jsonnet можно найти на https://github.com/ksonnet/ksonnet-lib.

17.5.5 Использование непрерывной интеграции и непрерывной доставки (CI/CD)

В двух разделах мы затронули вопрос автоматизации развертывания ресурсов Kubernetes, но вы можете настроить полный конвейер CI/CD для сборки двоичных файлов приложений, образов контейнеров и манифестов ресурсов, а затем их развертывания в одном или нескольких кластерах Kubernetes.

Данной теме посвящено достаточно много интернет-ресурсов. Здесь я хотел бы указать вам конкретно на проект Fabric8 (http://fabric8.io), который представляет собой интегрированную платформу разработки для Kubernetes. Она включает в себя Jenkins, хорошо известную систему автоматизации с открытым исходным кодом и различные другие инструменты для обеспечения полного конвейера CI/CD для разработки, развертывания и управления микросервисами на Kubernetes.

Если вы хотите создать свое собственное решение, я также предлагаю взглянуть на одну из онлайн-лабораторий облачной платформы Google, которая рассказывает об этой теме. Она доступна по адресу https://github.com/GoogleCloudPlatform/continuous-deployment-on-kubernetes.

17.6 Резюме

Надеемся, что информация в этой главе дала вам еще более глубокое понимание того, как Kubernetes работает и помогает вам собирать приложения, которые чувствуют себя как рыба в воде при развертывании в кластере Kubernetes. Целью этой главы было:

- показать, каким образом все ресурсы, описанные в этой книге, объединяются, чтобы представлять типичное приложение, работающее в Kubernetes;
- побудить вас задуматься о разнице между приложениями, которые редко перемещаются между машинами, и приложениями, работающими как модули, которые перемещаются гораздо чаще;

- помочь понять, что ваши многокомпонентные приложения (или микросервисы, если хотите) не должны полагаться на определенный порядок запуска;
- познакомить с контейнерами инициализации, которые можно использовать для инициализации модуля или задержки запуска главных контейнеров модуля до тех пор, пока не будет выполнено предварительное условие;
- познакомить вас с обработчиками жизненного цикла контейнера и тем, когда их использовать;
- дать более глубокое понимание последствий распределенной природы компонентов Kubernetes и ее возможной модели согласованности;
- познакомить с тем, как делать так, чтобы ваши приложения завершали свою работу должным образом, не прерывая клиентские подключения;
- дать вам несколько небольших советов о том, как сделать ваши приложения проще в управлении, сохраняя небольшие размеры образов, добавляя аннотации и многомерные метки во все ваши ресурсы и упрощая понимание того, почему приложение было терминировано;
- научить вас разрабатывать приложения Kubernetes и запускать их локально или в Minikube, перед тем как развертывать их в настоящем многоузловом кластере.

В следующей и последней главе мы узнаем, как расширять Kubernetes с помощью собственных индивидуально определяемых объектов API и контроллеров и как это делают другие, для того чтобы поверх Kubernetes создавать полные решения в парадигме платформы как службы.

Глава 18

Расширение системы Kubernetes

Эта глава посвящена:

- добавлению своих собственных объектов в Kubernetes;
- созданию контроллера для своего собственного объекта;
- добавлению своих собственных серверов API;
- саморезервированию служб с помощью каталога служб Kubernetes Service Catalog;
- контейнерной платформе OpenShift провайдера открытых решений Red Hat;
- системе Deis Workflow и менеджеру пакетов Helm.

Вы почти закончили. В заключение мы рассмотрим, каким образом можно определять свои собственные объекты API и создавать контроллеры для этих объектов. Мы также рассмотрим, как другие разработчики расширяли систему Kubernetes и строили поверх нее решения в парадигме платформы как службы.

18.1 Определение своих собственных объектов API

В этой книге вы познакомились с объектами API, предоставляемыми экосистемой Kubernetes, и с тем, как они используются для построения прикладных систем. В настоящее время пользователи Kubernetes в основном применяют только эти объекты, хотя они представляют собой относительно низкоуровневые общие понятия.

По мере развития экосистемы Kubernetes вы будете видеть все больше и больше объектов высокого уровня, которые будут гораздо более специализи-

рованными, чем ресурсы, которые Kubernetes поддерживает сегодня. Вместо того чтобы работать с развертываниями, службами, словарями конфигурации и т. д., вы будете создавать и управлять объектами, представляющими целые приложения или программные службы. Пользовательский контроллер будет наблюдать за этими высокоуровневыми объектами и создавать на их основе низкоуровневые объекты. Например, для запуска брокера сообщений в кластере Kubernetes достаточно создать экземпляр ресурса очереди Queue, а все необходимые секреты, развертывания и службы будут созданы пользовательским контроллером очереди. В Kubernetes уже предусмотрено добавление пользовательских ресурсов.

18.1.1 Знакомство с определениями CustomResourceDefinition

Чтобы определить новый тип ресурса, достаточно отправить объект CustomResourceDefinition (CRD) на сервер API Kubernetes. Объект CustomResourceDefinition – это описание своего собственного типа ресурса. После отправки CRD пользователи могут создавать экземпляры своего собственного ресурса путем отправки манифестов JSON или YAML на сервере API, как и в случае с любым другим ресурсом Kubernetes.

> **ПРИМЕЧАНИЕ.** До Kubernetes 1.7 задаваемые пользователями ресурсы определялись с помощью объектов ThirdPartyResource, которые были похожи на определения CustomResourceDefinition, но были удалены в версии 1.8.

Создание объекта CustomResourceDefinition, в результате которого пользователи смогут создавать объекты нового типа, не является полезным функциональным средством, если эти объекты не приводят к тому, что что-то происходит в кластере. Каждый объект CRD обычно также имеет связанный с ним контроллер (активный компонент, выполняющий что-то на основе пользовательских объектов), так же, как и все ключевые ресурсы Kubernetes, которые имеют связанный с ними контроллер, как было объяснено в главе 11. По этой причине, чтобы правильно показать, что именно определения CustomResourceDefinition позволяют делать, кроме добавления экземпляров пользовательского объекта, должен также быть развернут контроллер. Вы сделаете это в следующем далее примере.

Знакомство с примером объекта CustomResourceDefinition

Давайте представим, что вы хотите позволить пользователям вашего кластера Kubernetes запускать статические веб-сайты как можно проще, без необходимости иметь дело с модулями, службами и другими ресурсами Kubernetes. Вы хотите добиться, чтобы пользователи создавали объекты типа Website, которые не содержат ничего, кроме имени веб-сайта и источника, из которого должны быть получены файлы веб-сайта (HTML, CSS, PNG и др.). В качестве источника этих файлов вы будете использовать репозиторий Git.

Когда пользователь создает экземпляр объекта Website, вы хотите, чтобы система Kubernetes разворачивала новый веб-серверный модуль и предоставляла к нему доступ через службу, как показано на рис. 18.1.

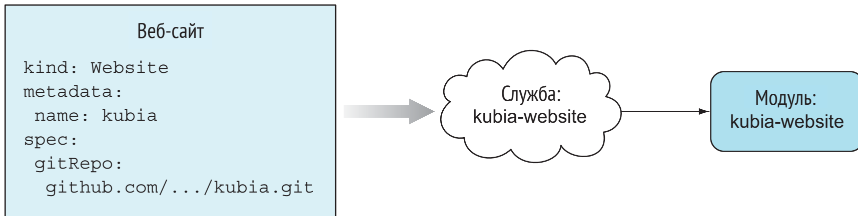


Рис. 18.1. Каждый объект веб-сайта Website должен привести к созданию службы и модуля сервера HTTP

Для того чтобы создать ресурс Website, необходимо, чтобы пользователи отправляли манифесты примерно следующего содержания, показанного в приведенном ниже листинге.

Листинг 18.1. Воображаемый ваш собственный ресурс imaginary-kubia-website.yaml

```
kind: Website            <- Вид пользовательского объекта
metadata:
  name: kubia            <- Имя веб-сайта (используется для именования
                            результирующей службы и модуля)
spec:
  gitRepo: https://github.com/luksa/kubia-website-example.git   <- Репозиторий Git,
                                                                   содержащий файлы
                                                                   веб-сайта
```

Как и все другие ресурсы, ваш ресурс содержит поле `kind` и `metadata.name`, как и большинство ресурсов, также содержит секцию спецификации `spec`. Он содержит одно поле под названием `gitRepo` – вы можете выбрать любое имя) – оно конкретизирует репозиторий Git, содержащий файлы веб-сайта. Вам также потребуется включить поле `apiVersion`, но вы еще не знаете, какое значение оно должно иметь для своих собственных ресурсов.

Если вы попробуете разместить этот ресурс в Kubernetes, то получите ошибку, потому что Kubernetes пока еще не знает, что это за объект Website:

```
$ kubectl create -f imaginary-kubia-website.yaml
error: unable to recognize "imaginary-kubia-website.yaml": no matches for
➥   /, Kind=Website
```

Прежде чем создавать экземпляры своего собственного объекта, необходимо сделать так, чтобы система Kubernetes их распознавала.

Создание объекта CustomResourceDefinition

Для того чтобы система Kubernetes принимала экземпляры ваших собственных ресурсов Website, необходимо отправить на сервер API определение CustomResourceDefinition, показанное в следующем ниже листинге.

Листинг 18.2. Манифест определения ресурса CustomResourceDefinition: website-crd.yaml

```
apiVersion: apiextensions.k8s.io/v1beta1      <- Полное имя вашего
kind: CustomResourceDefinition                   собственного объекта
metadata:
  name: websites.extensions.example.com      <- Определения CustomResourceDefinition
                                                принадлежат этой группе API и версии
spec:
  scope: Namespaced                          <- Вы хотите, чтобы ресурсы Website
                                                имели пространство имен
  group: extensions.example.com              <- Определить группу API
  version: v1                                   и версию ресурса Website
  names:
    kind: Website                            <- Необходимо указать разные формы
    singular: website                           имени своего собственного объекта
    plural: websites
```

После отправки дескриптора в систему Kubernetes она позволит создавать любое количество экземпляров пользовательского ресурса Website.

Объект CRD можно создать из файла website-crd.yaml, который имеется в архиве кода:

```
$ kubectl create -f website-crd-definition.yaml
customresourcedefinition "websites.extensions.example.com" created
```

Уверен, что вы задаетесь вопросом о длинном имени CRD. Почему бы не назвать его Website? Причина простая – чтобы предотвратить конфликты имен. Добавляя суффикс к имени CRD (который обычно включает имя организации, создавшей CRD), вы сохраняете уникальные имена объектов CRD. К счастью, длинное имя означает, что вам придется создавать ресурсы вашего сайта не как `kind: websites.extensions.example.com`, а как `kind: Website`. Это указано в свойстве `names.kind` ресурса CRD. Фрагмент `extensions.example.com` является группой API вашего ресурса.

Вы видели, как создание объектов Deployment требует выставления версии `apiVersion` в `apps/v1beta1` вместо `v1`. Фрагмент перед косой чертой – это группа API (развертывания принадлежат группе API `apps`), а фрагмент после нее – имя версии (в случае развертываний `v1beta1`). При создании экземпляров своего собственного ресурса Website необходимо для свойства `apiVersion` задать значение `extensions.example.com/v1`.

Создание экземпляра пользовательского ресурса

Учитывая то, что вы уже знаете, теперь вы создадите правильный YAML для вашего экземпляра ресурса Website. Соответствующий манифест YAML показан в следующем ниже листинге.

Листинг 18.3. Свой собственный ресурс Website: kubia-website.yaml

```
apiVersion: extensions.example.com/v1      <- Ваша собственная группа и версия API
```

```
kind: Website                          ◁ Этот манифест описывает
metadata:                                экземпляр ресурса Website
  name: kubia     ◁— Имя экземпляра Website
spec:
  gitRepo: https://github.com/luksa/kubia-website-example.git
```

Тип (`kind`) вашего ресурса – Website, а `apiVersion` состоит из группы и номера версии API, заданных в определении CustomResourceDefinition.

Теперь создайте объект Website:

```
$ kubectl create -f kubia-website.yaml
website "kubia" created
```

В отклике сообщается, что сервер API принял и сохранил ваш собственный объект Website. Давайте посмотрим, сможете ли вы его теперь извлечь.

Извлечение экземпляров пользовательского ресурса

Выведем список всех веб-сайтов в кластере:

```
$ kubectl get websites
NAME     KIND
kubia    Website.v1.extensions.example.com
```

Как и в случае с существующими ресурсами Kubernetes, вы можете создавать и выводить список экземпляров своих собственных ресурсов. Вы также можете использовать команду `kubectl describe` для просмотра сведений о своем собственном объекте или извлечения всего YAML с помощью `kubectl get`, как показано в следующем ниже листинге.

Листинг 18.4. Полное определение ресурса Website, полученное с сервера API

```
$ kubectl get website kubia -o yaml
apiVersion: extensions.example.com/v1
kind: Website
metadata:
  creationTimestamp: 2017-02-26T15:53:21Z
  name: kubia
  namespace: default
  resourceVersion: "57047"
  selfLink: /apis/extensions.example.com/v1/.../default/websites/kubia
  uid: b2eb6d99-fc3b-11e6-bd71-0800270a1c50
spec:
  gitRepo: https://github.com/luksa/kubia-website-example.git
```

Обратите внимание, что ресурс содержит все, что было в исходном определении YAML, и что Kubernetes инициализирует дополнительные поля метаданных так же, как и все другие ресурсы.

Удаление экземпляра пользовательского объекта

Очевидно, что, помимо создания и извлечения своих собственных объектов, вы можете также их удалять:

```
$ kubectl delete website kubia
website "kubia" deleted
```

> **ПРИМЕЧАНИЕ.** Вы удаляете экземпляр объекта Website, а не объект CRD Website. Вы также можете удалить сам объект CRD, но давайте пока на некоторое время от этого воздержимся, потому что в следующем разделе вы будете создавать дополнительные экземпляры объекта Website.

Давайте рассмотрим все, что вы сделали. Создание объекта CustomResourceDefinition позволяет сохранять, извлекать и удалять свои собственные объекты с помощью сервера API Kubernetes. Эти объекты пока ничего не делают. Для того чтобы заставить их что-то делать, вам нужно создать контроллер.

В общем случае создание своих собственных объектов, подобных этому, не всегда приводит к тому, что что-то происходит при создании объекта. Некоторые собственные объекты используются для хранения данных вместо использования более общего механизма, такого как словарь конфигурации ConfigMap. Приложения, работающие внутри модулей, могут запрашивать эти объекты на сервере API и считывать все, что в них хранится.

Но в данном случае мы поставили задачу, чтобы существование объекта Website привело к запуску веб-сервера, раздающего содержимое репозитория Git, на который ссылается этот объект. Посмотрим, как это сделать, дальше.

18.1.2 Автоматизация пользовательских ресурсов с помощью пользовательских контроллеров

Для того чтобы заставить объекты Website запускать веб-серверный модуль, доступ к которому предоставляется через службу, необходимо создать и развернуть контроллер Website, который будет наблюдать за тем, как сервер API создает объекты Website, а затем для каждого из них будет создавать службу и веб-серверный модуль.

Для того чтобы обеспечить управляемость модуля и его способность переживать аварийные сбои узла, контроллер создаст ресурс развертывания Deployment вместо непосредственного создания неуправляемого модуля. Работа контроллера резюмирована на рис. 18.2.

Я написал простенькую начальную версию контроллера, которая работает достаточно хорошо, для того чтобы показать в действии CRD и контроллер, но она далека от готовности к производственной среде, потому что слишком упрощена. Образ контейнера имеется на `docker.io/luksa/website-controller:latest`, исходный код – на `https://github.com/luksa/k8swebsite-controller`. Вместо того чтобы просматривать исходный код контроллера, я объясню, что он делает.

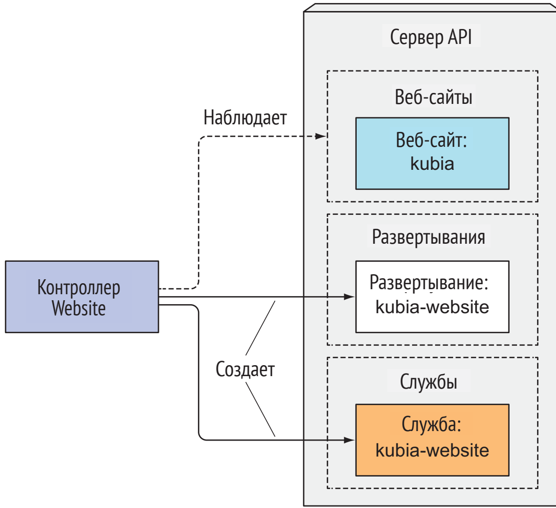


Рис. 18.2. Контроллер Website наблюдает за объектами Website и создает объекты Deployment и Service

Что делает контроллер Website

Сразу после запуска контроллер начинает просматривать объекты Website, запрашивая следующий ниже URL-адрес:

```
http://localhost:8001/apis/extensions.example.com/v1/websites?watch=true
```

Вы можете узнать хостнейм и порт – контроллер не подключается к серверу API напрямую, а вместо этого подключается к процессу `kubectl proxy`, который работает в побочном контейнере в том же модуле и действует как посредник на сервере API (мы рассмотрели шаблон посла в главе 8). Прокси перенаправляет запрос на сервер API, занимаясь и шифрованием TLS, и аутентификацией (см. рис. 18.3).

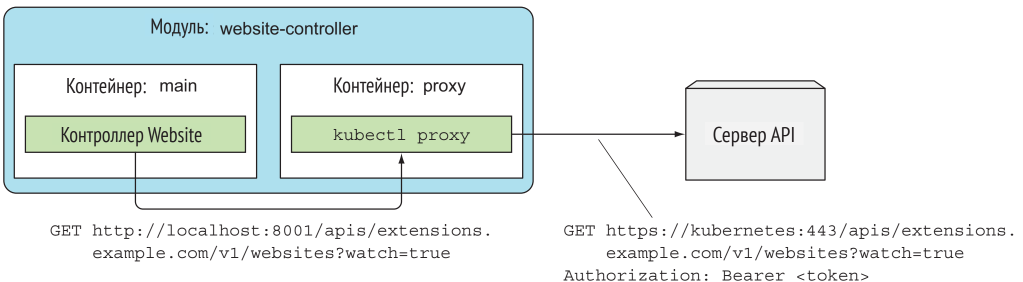


Рис. 18.3. Контроллер Website обменивается с сервером API через прокси (в контейнере-после)

Через соединение, открытое этим запросом HTTP GET, сервер API будет отправлять события наблюдения по каждому изменению в любом объекте Website.

Сервер API отправляет событие наблюдения `ADDED` всякий раз, когда создается новый объект Website. Когда контроллер получает такое событие, он

извлекает имя веб-сайта и URL-адрес репозитория Git из объекта Website, который он получил в событии наблюдения, и создает объект Deployment и Service, отправляя их манифесты JSON на сервер API.

Ресурс развертывания Deployment содержит шаблон двухконтейнерного модуля (рис. 18.4): один выполняет сервер Nginx, а другой – процесс git-sync, который синхронизирует локальный каталог с содержимым репозитория Git. Локальный каталог используется совместно с контейнером Nginx посредством тома `emptyDir` (вы делали что-то похожее в главе 6, но вместо синхронизации локального каталога с репозиторием Git вы использовали том `gitRepo` для скачивания содержимого репозитория Git при запуске модуля; впоследствии содержимое тома с репозиторием Git не синхронизировалось). Служба является службой `NodePort`, которая предоставляет доступ к вашему веб-серверному модулю через случайный порт на каждом узле (на всех узлах используется одинаковый порт). Когда модуль создается объектом развертывания Deployment, клиенты могут получать доступ к веб-сайту через порт узла.

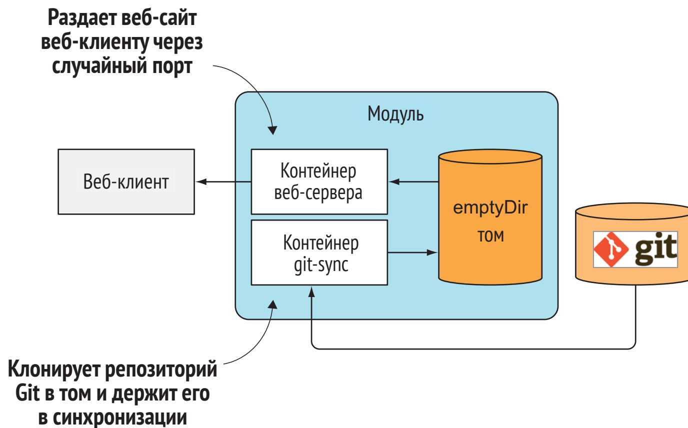


Рис. 18.4. Модуль обслуживает веб-сайт, указанный в объекте Website

Сервер API также отправляет событие наблюдения `DELETED`, когда экземпляр ресурса Website удаляется. После получения события контроллер удаляет созданные ранее ресурсы развертывания Deployment и службы Service. Как только пользователь удаляет экземпляр Website, контроллер завершает работу и удаляет веб-сервер, обслуживающий этот веб-сайт.

> **ПРИМЕЧАНИЕ.** Мой упрощенный контроллер не реализован должным образом. То, как он наблюдает за объектами API, не гарантирует, что он не пропустит отдельные события наблюдения. Правильный способ наблюдать за объектами через сервер API состоял бы не только в наблюдении за ними, но и в периодическом повторном внесении в список всех объектов для проверки, были какие-либо события наблюдения пропущены или нет.

Запуск контроллера в качестве модуля

Во время разработки я запустил контроллер на своем локальном ноутбуке, предназначенном для разработки, и использовал локально работающий процесс `kubectl proxy` (не работающий как модуль) в качестве посла на сервере API Kubernetes. Это позволило мне вести быструю разработку, потому что мне не нужно было создавать образ контейнера после каждого изменения исходного кода, а затем запускать его внутри Kubernetes.

Когда все готово к развертыванию контроллера в производственной среде, лучше всего запускать контроллер внутри самой системы Kubernetes, как это делается со всеми другими ключевыми контроллерами. Для того чтобы запустить контроллер в Kubernetes, его можно развернуть с помощью ресурса развертывания Deployment. Ниже приведен пример такого развертывания.

Листинг 18.5. Развертывание контроллера веб-сайта: website-controller.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: website-controller
spec:
  replicas: 1                                  <- Вы запустите единственную копию контроллера
  template:
  metadata:
    name: website-controller
    labels:
      app: website-controller
  spec:
    serviceAccountName: website-controller     <- Он будет работать под специальной учетной записью ServiceAccount
    containers:
    - name: main
      image: luksa/website-controller          <- Два контейнера: главный контейнер и побочный прокси
    - name: proxy
      image: luksa/kubectl-proxy:1.6.2
```

Как видите, объект Deployment развертывает одну реплику двухконтейнерного модуля. Один контейнер запускает ваш контроллер, в то время как другой является контейнером-посредником, используемым для более простого обмена с сервером API. Данный модуль работает под собственной специальной учетной записью ServiceAccount, поэтому ее необходимо создать перед развертыванием контроллера:

```
$ kubectl create serviceaccount website-controller
serviceaccount "website-controller" created
```

Если в кластере активировано управление ролевым доступом (RBAC), то Kubernetes не позволит контроллеру просматривать ресурсы Website или соз-

давать развертывания Deployment или службы Service. Для того чтобы это стало возможным, вам потребуется привязать учетную запись `website-controller` к кластерной роли `cluster-admin`, создав привязку ClusterRoleBinding, как здесь:

```
$ kubectl create clusterrolebinding website-controller
➥  --clusterrole=cluster-admin
➥ --serviceaccount=default:website-controller
clusterrolebinding "website-controller" created
```

Имея в распоряжении учетную запись ServiceAccount и привязку кластерной роли ClusterRoleBinding, вы можете развернуть объект Deployment контроллера.

Контроллер в действии

Теперь, когда контроллер работает, снова создайте ресурс Website `kubia`:

```
$ kubectl create -f kubia-website.yaml
website "kubia" created
```

Теперь давайте проверим журналы событий контроллера (показанные в следующем ниже списке), чтобы убедиться, что он получил событие наблюдения.

Листинг 18.6. Вывод журналов контроллера веб-сайта

```
$ kubectl logs website-controller-2429717411-q43zs -c main
2017/02/26 16:54:41 website-controller started.
2017/02/26 16:54:47 Received watch event: ADDED: kubia: https://github.c...
2017/02/26 16:54:47 Creating services with name kubia-website in namespa...
2017/02/26 16:54:47 Response status: 201 Created
2017/02/26 16:54:47 Creating deployments with name kubia-website in name...
2017/02/26 16:54:47 Response status: 201 Created
```

Журналы показывают, что контроллер получил событие `ADDED` и что он создал службу и развертывание для Website `kubia-website`. Сервер API ответил откликом `201 Created`, который означает, что два ресурса должны теперь существовать. Давайте проверим, что развертывание, служба и результирующий модуль были созданы. В следующем ниже листинге перечислены все развертывания, службы и модули.

Листинг 18.7. Объекты Deployment, Service и Pod, созданные для kubia-website

```
$ kubectl get deploy,svc,po
NAME                    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/kubia-website    1         1         1            1           4s
```

```
deploy/website-controller  1        1        1           1          5m

NAME                CLUSTER-IP    EXTERNAL-IP  PORT(S)       AGE
svc/kubernetes      10.96.0.1     <none>       443/TCP       38d
svc/kubia-website   10.101.48.23  <nodes>      80:32589/TCP  4s

NAME                                   READY  STATUS   RESTARTS  AGE
po/kubia-website-1029415133-rs715      2/2    Running  0         4s
po/website-controller-1571685839-qzmg6 2/2    Running  1         5m
```

Все на месте. Служба `kubia-website`, через которую вы можете обращаться к вашему сайту, доступна на порту `32589` на всех узлах кластера. Вы можете получать к нему доступ с помощью браузера. Потрясающе, правда?

Пользователи кластера Kubernetes теперь могут развертывать статические веб-сайты за считанные секунды, не зная ничего о модулях, службах или любых других ресурсах Kubernetes, кроме пользовательского ресурса Website.

Совершенно очевидно, что у вас еще есть место для совершенствования. Контроллер может, например, наблюдать за объектами службы и, как только порт узла назначается, записывать в секцию `status` самого экземпляра ресурса Website URL-адрес, на котором веб-сайт доступен. Или же он мог бы для каждого веб-сайта создавать объект входа Ingress. Я оставлю реализацию этого дополнительного функционала вам в качестве упражнения.

18.1.3 Валидация пользовательских объектов

Возможно, вы заметили, что в описаниях CustomResourceDefinition ресурса Website вы не указали никакой схемы валидации. Пользователи могут включать в YAML своего объекта Website любое поле, которое они захотят. Сервер API не проверяет допустимость содержимого YAML (за исключением обычных полей, таких как `apiVersion`, `kind` и `metadata`), поэтому пользователи могут создавать недопустимые объекты Website (например, без поля `gitRepo`).

Можно ли добавить в контроллер валидацию и предотвратить принятие недопустимых объектов сервером API? Ответ – нет, потому что сервер API сначала сохраняет объект, а затем возвращает клиенту (`kubectl`) отклик об успешном выполнении и только потом уведомляет всех наблюдателей (контроллер является одним из них). То, что контроллер действительно может сделать, – так это проверить допустимость объекта, когда он получает его в событии наблюдения, и, если объект является недопустимым, написать сообщение об ошибке объекту Website (путем обновления объекта с помощью нового запроса к серверу API). Пользователь не будет уведомлен об ошибке автоматически. Он должен заметить сообщение об ошибке, запросив у сервера API объект Website. Если пользователь этого не сделает, то у него нет способа узнать, является объект допустимым или нет.

Совершенно очевидно, что такая ситуация не идеальна. Вы хотели бы, чтобы сервер API немедленно выполнял валидацию объекта и отклонял недопустимые объекты. Валидация пользовательских объектов была введена в

Kubernetes версии 1.8 в качестве альфа-функции. Для того чтобы сервер API проверял допустимость пользовательских объектов, вам нужно активировать функционал `customResourceValidation` на сервере API и задать схему JSON в описании CRD.

18.1.4 Предоставление пользовательского сервера API для пользовательских объектов

Более оптимальный способ добавить поддержку своих собственных объектов в Kubernetes – реализовать собственный сервер API и дать клиентам возможность обращаться к нему напрямую.

Знакомство с агрегацией сервера API

В Kubernetes версии 1.7 можно интегрировать свой собственный сервер API с главным сервером API Kubernetes посредством агрегации серверов API. Изначально сервер API Kubernetes был единственным монолитным компонентом. Начиная с версии 1.7 Kubernetes обеспечивается доступ ко множеству агрегированных серверов API в одном месте. Клиенты могут подключаться к агрегированному API и прозрачно перенаправлять свои запросы на соответствующий сервер API. Благодаря этому клиент даже не будет знать, что за кулисами различные объекты обрабатываются несколькими серверами API. Даже главный сервер API Kubernetes может в конечном итоге быть разделен на несколько небольших серверов API и через агрегатор выставлен как один сервер, как показано на рис. 18.5.

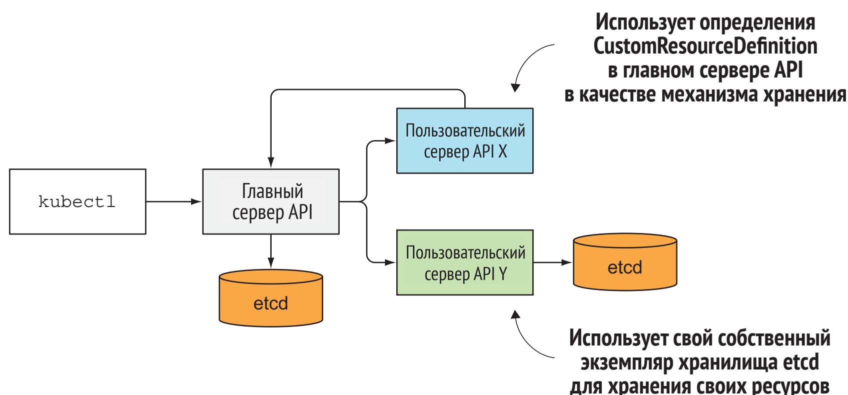


Рис. 18.5. Агрегация сервера API

В вашем случае можно создать сервер API, отвечающий за обработку объектов Website. Он может проверять допустимость этих объектов точно так же, как их проверяет главный сервер API Kubernetes. Вам больше не нужно создавать описание CRD для представления этих объектов, потому что тип объекта Website будет реализован непосредственно на своем собственном сервере API.

Как правило, каждый сервер API отвечает за хранение своих ресурсов. Как показано на рис. 18.5, он может либо запускать свой экземпляр хранилища

etcd (или весь кластер etcd), либо хранить свои ресурсы в хранилище etcd главного сервера API путем создания экземпляров CRD на главном сервере API. В этом случае, прежде чем создавать экземпляры CRD, как это было в примере выше, необходимо сначала создать объект CRD.

Регистрация пользовательского сервера API

Для того чтобы добавить свой собственный сервер API в кластер, необходимо развернуть его как модуль и предоставить доступ к нему через службу. Затем, чтобы интегрировать его в главный сервер API, вы развернете манифест YAML, описывающий ресурс APIService, подобный тому, который приведен в следующем ниже листинге.

Листинг 18.8. Определение YAML службы APIservice

```
apiVersion: apiregistration.k8s.io/v1beta1      <-- Это ресурс APIService
kind: APIService
metadata:
  name: v1alpha1.extensions.example.com
spec:
  group: extensions.example.com                 <-- Группа API, за которую отвечает этот сервер API
  version: v1alpha1                             <-- Поддерживаемая версия API
  priority: 150
  service:
    name: website-api                           <-- Служба, через которую предоставляется
    namespace: default                              доступ к своему собственному серверу API
```

После создания ресурса APIService из приведенного выше листинга клиентские запросы, отправляемые на главный сервер API, содержащий любой ресурс из группы API `extensions.example.com` и его версии `v1alpha1`, будут перенаправлены на модули с пользовательскими серверами API, предоставляемые через службу `website-api`.

Создание пользовательских клиентов

Вы можете создавать свои собственные ресурсы из файлов YAML с помощью обычного клиента `kubectl`. Вместе с тем, для того чтобы сделать развертывание своих собственных объектов еще проще, в дополнение к предоставлению пользовательского сервера API вы также можете создавать свой собственный инструмент командной строки CLI. Он позволит вам добавлять отдельные команды для работы с этими объектами, подобно тому, как `kubectl` позволяет создавать секреты Secret, развертывания Deployment и другие ресурсы посредством ресурсно-специфических команд, таких как `kubectl create secret` или `kubectl create deployment`.

Как уже отмечалось ранее, пользовательские API-серверы, агрегация серверов API и другие функциональные средства, связанные с расширением Kubernetes, в настоящее время интенсивно разрабатываются, поэтому они могут измениться после публикации данной книги. Для того чтобы полу-

чить актуальную информацию по этому вопросу, обратитесь к репозиториям Kubernetes GitHub по адресу http://github.com/kubernetes.

18.2 Расширение Kubernetes с помощью каталога служб Kubernetes (Kubernetes Service Catalog)

Одним из первых дополнительных серверов API, которые будут добавлены в Kubernetes посредством агрегации серверов API, является сервер API каталога служб (Service Catalog API server). В сообществе Kubernetes каталог служб является горячей темой, так что вы, возможно, захотите узнать о нем подробнее.

В настоящее время, для того чтобы модуль потреблял службу (здесь я использую этот термин в общем понимании, не по отношению к ресурсам службы (Service); например, служба базы данных включает в себя все необходимое, чтобы позволить пользователям применять базу данных в своем приложении), кто-то должен развернуть предоставляющие службу модули, ресурс службы и, возможно, секрет, чтобы клиентский модуль мог использовать его для аутентификации в службе. Этот пользователь обычно является тем же пользователем, который развертывает клиентский модуль, или если для развертывания этих типов общих служб выделена группа, то пользователь должен подать заявку и дождаться, пока команда зарезервирует службу. Это означает, что пользователь должен либо создавать манифесты для всех компонентов службы, знать, где найти существующий набор манифестов, знать, как настроить его должным образом, и развернуть его вручную, либо ждать до тех пор, пока это сделает другая команда разработчиков.

Однако система Kubernetes должна быть простой в обращении системой самообслуживания. В идеале пользователи, приложения которых требуют определенной службы (например, веб-приложение, требующее бэкенд-базу данных), должны иметь возможность сообщать Kubernetes «Эй, мне нужна база данных PostgreSQL. Пожалуйста, предоставь ее мне и сообщи, где и как я смогу к ней подключиться». Это вскоре станет возможным благодаря каталогу служб Kubernetes.

18.2.1 Знакомство с каталогом служб

Как следует из названия, этот каталог представляет собой каталог служб. Пользователи могут самостоятельно просматривать каталог и создавать экземпляры перечисленных в каталоге служб без необходимости работать с модулями, службами, словарями конфигурации и другими ресурсами, необходимыми для запуска службы. Вы сразу поймете, что это похоже на то, что вы делали со своим собственным ресурсом Website.

Вместо добавления своих собственных ресурсов в сервер API для каждого типа службы в каталоге служб представлены следующие четыре универсальных API-ресурса:

- брокер кластерных служб ClusterServiceBroker, описывающий (внешнюю) систему, которая может создавать службы;
- класс кластерной службы ClusterServiceClass, описывающий тип службы, которую можно создавать;
- экземпляр службы ServiceInstance, то есть один экземпляр службы, который был создан;
- привязка службы ServiceBinding, которая представляет собой привязку между набором клиентов (модулями) и экземпляром службы ServiceInstance.

Взаимосвязь между этими четырьмя ресурсами показана на рис. 18.6 и поясняется в нижеследующих абзацах.

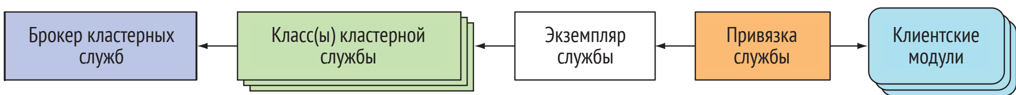


Рис. 18.6. Взаимосвязь между API-ресурсами каталога служб

В двух словах администратор кластера создает ресурс ClusterServiceBroker для каждого компонента брокера служб, чьи службы он хочет сделать доступными в кластере. Затем Kubernetes запрашивает у брокера список служб, которые тот может предоставить, и создает ресурс ClusterServiceClass для каждого из них. Когда пользователю требуется зарезервировать службу, он создает ресурс с экземпляром ServiceInstance, а затем привязку ServiceBinding, чтобы привязать этот экземпляр к своим модулям. В эти модули затем внедряется секрет, который содержит все необходимые учетные данные и другие данные, необходимые для подключения к зарезервированному экземпляру ServiceInstance.

Архитектура системы каталога служб показана на рис. 18.7.

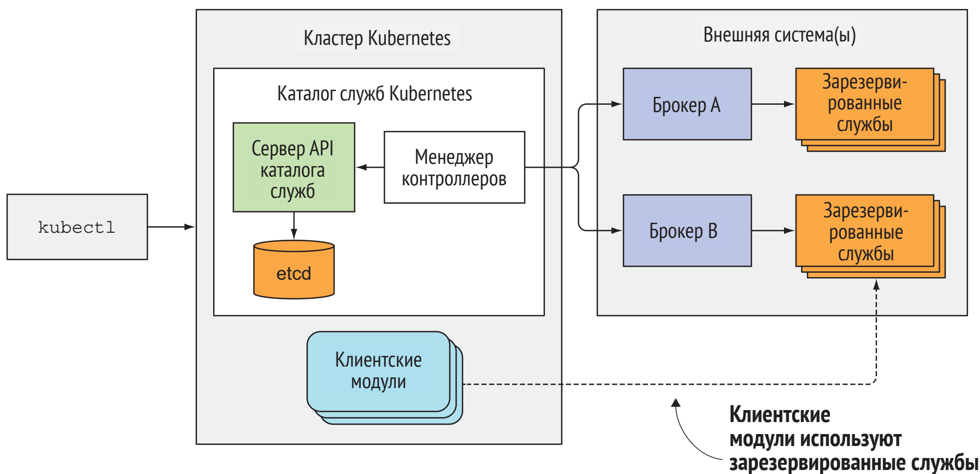


Рис. 18.7. Архитектура каталога служб

Показанные на рисунке компоненты описаны в следующих далее разделах.

18.2.2 Знакомство с сервером API каталога служб и менеджером контроллеров

Подобно ядру системы Kubernetes, каталог служб представляет собой распределенную систему, состоящую из трех компонентов:

- сервер API каталога служб;
- хранилище etcd;
- менеджер контроллеров, в котором работают все контроллеры.

Четыре ранее представленных ресурса, связанных с каталогом служб, создаются путем публикации манифестов YAML/JSON на сервере API. Затем он сохраняет их в собственном экземпляре хранилища etcd или использует определения CustomResourceDefinition на главном сервере API в качестве альтернативного механизма хранения (в этом случае дополнительный экземпляр etcd не требуется).

Контроллеры, работающие в менеджере контроллеров, делают что-то с этими ресурсами. Они, безусловно, обмениваются с сервером API каталога служб, точно так же, как это делают ключевые контроллеры Kubernetes, взаимодействуя с главным сервером API. Эти контроллеры сами не предоставляют запрашиваемые службы. Они оставляют это на усмотрение внешних брокеров служб, которые регистрируются путем создания ресурсов ServiceBroker в API каталога служб.

18.2.3 Знакомство с брокерами служб и API OpenServiceBroker

Администратор кластера может зарегистрировать одного или нескольких внешних брокеров служб ServiceBroker в каталоге служб. Каждый брокер должен реализовывать API OpenServiceBroker.

Знакомство с API OpenServiceBroker

Каталог служб обращается к брокеру через этот API. Данный API относительно прост. Это API REST, предоставляющий следующие операции:

- получение списка служб с помощью `GET/v2/catalog`;
- резервирование экземпляра службы (`PUT/v2/service_instances/:id`);
- обновление экземпляра службы (`PATCH/v2/service_instances/:id`);
- привязывание экземпляра службы (`PUT/v2/service_instances/:id/service_bindings/:binding_id`);
- отвязывание экземпляра (`DELETE/v2/service_instances/:id/service_bindings/:binding_id`);
- дерезервирование экземпляра службы (`DELETE/v2/service_instances/:id`).

Спецификацию API OpenServiceBroker можно найти по адресу https://github.com/openservicebrokerapi/servicebroker.

Регистрация брокеров в каталоге служб

Администратор кластера регистрирует брокера путем публикации манифеста ресурса ServiceBroker в API каталога служб, как показано в следующем ниже листинге.

Листинг 18.9. Манифест брокера ClusterServiceBroker: database-broker.yaml

```
apiVersion: servicecatalog.k8s.io/v1alpha1    <- Вид ресурса, группа
kind: ClusterServiceBroker                        и версия API
metadata:
  name: database-broker                        <- Имя этого брокера
spec:
  url: http://database-osbapi.myorganization.org   <- Где каталог служб может связаться
                                                      с брокером (его URL-адрес
                                                      API OpenServiceBroker [OSB])
```

В данном листинге описывается воображаемый брокер, который может предоставлять базы данных различных типов. После того как администратор создает ресурс, контроллер ClusterServiceBroker в менеджере контроллеров каталога служб подключается к указанному в ресурсе URL-адресу для получения списка служб, которые может предоставить этот брокер.

После того как каталог служб получит список служб, для каждой из них он создает ресурс ClusterServiceClass. Каждый ресурс ClusterServiceClass описывает один тип службы, который может быть зарезервирован (примером ресурса ClusterServiceClass является «база данных PostgreSQL»). С каждым классом ClusterServiceClass связан один или несколько планов обслуживания. Это позволяет пользователю выбрать уровень службы, в котором он нуждается (например, база данных ClusterServiceClass может обеспечивать «бесплатный» план, где размер базы данных лимитирован и хранилищем является вращающийся диск, и «премиальный» план с неограниченным размером базы данных и SSD-накопителем).

Вывод списка служб, имеющихся в кластере

Как показано в следующем ниже листинге, пользователи кластера Kubernetes могут получить список всех служб, которые могут быть зарезервированы в кластере с помощью команды `kubectl get serviceclasses`.

Листинг 18.10. Список классов ClusterServiceClass в кластере

```
$ kubectl get clusterserviceclasses
NAME                KIND
postgres-database   ClusterServiceClass.v1alpha1.servicecatalog.k8s.io
mysql-database      ServiceClass.v1alpha1.servicecatalog.k8s.io
mongodb-database    ServiceClass.v1alpha1.servicecatalog.k8s.io
```

В данном списке показаны классы кластерных служб ClusterServiceClass для служб, которые может предоставить воображаемый брокер базы данных. Вы

можете сравнить классы ClusterServiceClass с классами хранения StorageClass, о которых мы говорили в главе 6. Классы хранения StorageClass позволяют выбирать тип хранилища, который вы хотите использовать в своих модулях, в то время как классы ClusterServiceClass позволяют выбирать тип службы.

Вы можете просмотреть подробные сведения об одном из классов ClusterServiceClass, получив его YAML. Соответствующий пример показан в следующем ниже листинге.

Листинг 18.11. Определение класса кластерных служб ClusterServiceClass

```
$ kubectl get serviceclass postgres-database -o yaml
apiVersion: servicecatalog.k8s.io/v1alpha1
bindable: true
brokerName: database-broker          <-- Этот класс ClusterServiceClass
description: A PostgreSQL database       предоставлен брокером базы
kind: ClusterServiceClass                данных database-broker
metadata:
  name: postgres-database
  ...
planUpdatable: false
plans:
- description: A free (but slow) PostgreSQL instance
  name: free                                           <-- Бесплатный план
  osbFree: true                                            для этой службы
  ...
- description: A paid (very fast) PostgreSQL instance
  name: premium                                        <-- Платный план
  osbFree: false
  ...
```

Класс кластерных служб ClusterServiceClass в приведенном выше списке содержит два плана – бесплатный план `free` и премиальный план `premium`. Вы видите, что этот класс ClusterServiceClass предоставлен брокером `database-broker`.

18.2.4 Резервирование и использование службы

Давайте представим, что развертываемые вами модули должны использовать базу данных. Вы проинспектировали список располагаемых классов ClusterServiceClass и решили использовать класс ClusterServiceClass `postgres-database` со свободным планом (`free`).

Резервирование экземпляра службы ServiceInstance

Для того чтобы для вас была зарезервирована база данных, от вас требуется только создать ресурс ServiceInstance. Это показано в следующем ниже листинге.

Листинг 18.12. Манифест экземпляра службы: database-instance.yaml

```
apiVersion: servicecatalog.k8s.io/v1alpha1
kind: ServiceInstance
metadata:
  name: my-postgres-db
spec:
  clusterServiceClassName: postgres-database
  clusterServicePlanName: free
  parameters:
    init-db-args: --data-checksums
```

Вы даете этому экземпляру имя

Класс ServiceClass и план, который вы хотите

Дополнительные параметры, передаваемые брокеру

Вы создали экземпляр ServiceInstance с именем `my-postgres-db` (это будет имя развертываемого ресурса) и указали класс ClusterServiceClass и выбранный план. Вы также указываете параметр, специфичный для каждого брокера и класса ClusterServiceClass. Представим, что вы отыскали возможные параметры в документации брокера.

Как только вы создадите этот ресурс, каталог служб свяжется с брокером, которому принадлежит класс ClusterServiceClass, и попросит службу зарезервировать его. Он передаст выбранные имя класса ClusterServiceClass и план, а также все указанные вами параметры.

Далее он полностью зависит от брокера в том, чтобы знать, что делать с этой информацией. В вашем случае брокер баз данных, вероятно, запустит где-то новый экземпляр базы данных PostgreSQL – не обязательно в том же кластере Kubernetes или даже в Kubernetes вообще. Он может запустить виртуальную машину и запустить там базу данных. Каталогу служб все равно, и то же самое касается и пользователя, который запрашивает службу.

Как показано в следующем ниже листинге, вы можете проверить успешность резервирования службы, проинспектировав секцию `status` созданного вами экземпляра службы `my-postgres-db`.

Листинг 18.13. Проверка статуса экземпляра службы

```
$ kubectl get instance my-postgres-db -o yaml
apiVersion: servicecatalog.k8s.io/v1alpha1
kind: ServiceInstance
...
status:
  asyncOpInProgress: false
  conditions:
  - lastTransitionTime: 2017-05-17T13:57:22Z
    message: The instance was provisioned successfully
    reason: ProvisionedSuccessfully
    status: "True"
    type: Ready
```

База данных была успешно зарезервирована

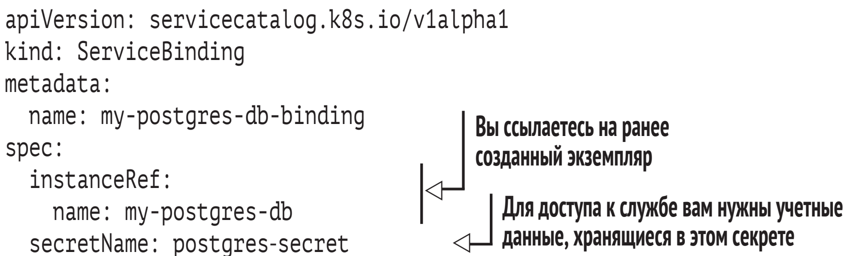
Она готова к использованию

Экземпляр базы данных теперь где-то работает, но как ее использовать в своих модулях? Для того чтобы это сделать, вам нужно ее привязать.

Привязывание экземпляра службы

Для того чтобы использовать зарезервированный экземпляр ServiceInstance в модулях, создайте ресурс ServiceBinding, как показано в следующем ниже листинге.

Листинг 18.14. ПривязкаServiceBinding: my-postgres-db-binding.yaml




```
apiVersion: servicecatalog.k8s.io/v1alpha1
kind: ServiceBinding
metadata:
  name: my-postgres-db-binding
spec:
  instanceRef:
    name: my-postgres-db
  secretName: postgres-secret
```

Данный листинг показывает, что вы определяете ресурс привязки ServiceBinding под названием `my-postgresdb-binding`, в котором вы ссылаетесь на экземпляр созданной вами ранее службы `my-postgres-db`. Вы также указываете имя секрета. Вы хотите, чтобы каталог служб поместил в секрет под названием `postgres-secret` все необходимые учетные данные для доступа к экземпляру службы. Но где в своих модулях следует привязывать экземпляр ServiceInstance. Вообще-то, нигде.

В настоящее время каталог служб еще не позволяет внедрять модули с учетными данными экземпляра ServiceInstance. Это станет возможным, когда станет доступной новый функционал Kubernetes под названием PodPresets. До тех пор вы можете выбирать имя для секрета там, где вы хотите хранить учетные данные, и монтировать этот секрет в ваши модули вручную.

При отправке ресурса ServiceBinding из предыдущего листинга на сервер API каталога служб контроллер снова свяжется с брокером базы данных и создаст привязку для ранее зарезервированного экземпляра ServiceInstance. Брокер откликается списком учетных данных и другими данными, необходимыми для подключения к базе данных. Каталог служб создает новый секрет с именем, указанным в ресурсе ServiceBinding, и сохраняет все эти данные в секрете.

Использование вновь созданного секрета в клиентских модулях

Секрет, созданный системой каталога служб, можно смонтировать в модули, чтобы те могли считывать его содержимое и использовать его для подключения к зарезервированному экземпляру службы (в данном примере к базе данных PostgreSQL). Секрет может выглядеть, как показано в следующем ниже листинге.

Листинг 18.15. Секрет, содержащий учетные данные для подключения к экземпляру службы

```
$ kubectl get secret postgres-secret -o yaml
apiVersion: v1
data:
  host: <base64-encoded hostname of the database>
  username: <base64-encoded username>
  password: <base64-encoded password>
kind: Secret
metadata:
  name: postgres-secret
  namespace: default
  ...
type: Opaque
```

Это то, что модуль должен использовать для подключения к службе базы данных

Поскольку имя секрета можно выбирать самостоятельно, вы можете разворачивать модули перед резервированием или привязкой службы. Как вы узнали в главе 7, модули не будут запущены до тех пор, пока такой секрет не будет существовать.

При необходимости можно создать несколько привязок для разных модулей. Брокер службы может использовать один и тот же набор учетных данных в каждой привязке, но лучше создавать новый набор учетных данных для каждого экземпляра привязки. Благодаря этому можно предотвратить ситуацию, когда модули будут использовать службы путем удаления ресурса ServiceBinding.

18.2.5 Отвязывание и дерезервирование

Если привязка службы больше не нужна, ее можно удалить так же, как и другие ресурсы:

```
$ kubectl delete servicebinding my-postgres-db-binding
servicebinding "my-postgres-db-binding" deleted
```

При этом контроллер каталога служб удаляет секрет и вызывает брокера для выполнения операции отмены привязки. Экземпляр службы (в вашем случае база данных PostgreSQL) все еще работает. Поэтому, если хотите, вы можете создать новую привязку ServiceBinding.

Но если экземпляр базы данных больше не нужен, следует также удалить ресурс экземпляра службы:

```
$ kubectl delete serviceinstance my-postgres-db
serviceinstance "my-postgres-db" deleted
```

Удаление ресурса ServiceInstance заставляет каталог служб выполнить операцию дерезервирования на брокере службы. Опять-таки, что именно это означает, зависит от брокера службы, но в вашем случае брокер должен за-

вершить работу экземпляра базы данных PostgreSQL, который он создал при резервировании экземпляра службы.

18.2.6 Что дает каталог служб

Как вы узнали, каталог служб позволяет поставщикам служб предоставлять доступ к этим службам в любом кластере Kubernetes путем регистрации брокера в этом кластере.

Например, я был связан с каталогом служб с самого начала и реализовал брокер, который делает тривиальной задачу резервирования систем обмена сообщениями и предоставляет к ним доступ модулям в кластере Kubernetes. Еще одна команда разработчиков внедрила брокера, который упрощает резервирование веб-служб Amazon.

В общем случае брокеры служб позволяют легко резервировать и предоставлять доступ к службам в Kubernetes и в ближайшее время сделают Kubernetes еще более удивительной платформой для развертывания ваших приложений.

18.3 Платформы, построенные поверх Kubernetes

Уверен, вы согласитесь, что Kubernetes – отличная система сама по себе. Учитывая, что она легко расширяется во всех ее компонентах, неудивительно, что компании, которые ранее разрабатывали свои собственные пользовательские платформы, теперь заново их реализуют поверх Kubernetes. Kubernetes, по сути, становится широко признанной основой для нового поколения предложений в парадигме платформы как службы (PaaS).

Среди наиболее известных систем PaaS, построенных на Kubernetes, являются Workflow компании Deis и OpenShift от провайдера открытых систем Red Hat. Мы сделаем краткий обзор обеих систем, чтобы дать вам представление о том, что они предлагают поверх всех этих удивительных вещей, которые уже предлагает экосистема Kubernetes.

18.3.1 Контейнерная платформа Red Hat OpenShift

Red Hat OpenShift – это платформа как служба, и поэтому она уделяет большое внимание удобству процесса разработки. Среди ее целей – обеспечение быстрой разработки приложений, а также простое развертывание, масштабирование и долгосрочное обслуживание этих приложений. Платформа OpenShift существует гораздо дольше, чем Kubernetes. Ее версии 1 и 2 были собраны с нуля и не имели ничего общего с Kubernetes, но, когда была анонсирована экосистема Kubernetes, компания Red Hat решила перестроить платформу OpenShift версии 3 с нуля – но на этот раз поверх экосистемы Kubernetes. Когда такая компания, как Red Hat, решает выбросить старую версию своего программного обеспечения и построить новую поверх существующей техно-

логии, такой как Kubernetes, всем должно быть ясно, насколько замечательной является экосистема Kubernetes.

Kubernetes автоматизирует развертывание и масштабирование приложений, в то время как платформа OpenShift также автоматизирует фактическую сборку образов приложений и их автоматическое развертывание, не требуя внедрять в ваш кластер решения непрерывной интеграции.

Платформа OpenShift также обеспечивает управление пользователями и группами, что позволяет запускать должным образом защищенный мультитенантный (многоклиентский) кластер Kubernetes, где отдельным пользователям разрешен доступ только к собственным пространствам имен Kubernetes, а приложения, работающие в этих пространствах имен, также полностью изолированы друг от друга по умолчанию.

Знакомство с дополнительными ресурсами в OpenShift

Платформа OpenShift обеспечивает несколько дополнительных объектов API в дополнение ко всем имеющимся в Kubernetes. Мы объясним их в следующих нескольких абзацах, чтобы предоставить вам хороший обзор того, что делает OpenShift и что она предоставляет.

Дополнительные ресурсы включают:

- пользователи User и группы Group;
- проекты Project;
- шаблоны Template;
- конфигурации сборок BuildConfig;
- конфигурации развертываний DeploymentConfig;
- потоки образов ImageStream;
- маршруты Route
- и другие.

Пользователи, группы и проекты

Мы уже отметили, что платформа OpenShift предоставляет пользователям надлежащую многопользовательскую среду. В отличие от экосистемы Kubernetes, которая не имеет объекта API для представления отдельного пользователя кластера (но имеет учетные записи ServiceAccount, которые представляют работающие в нем службы), OpenShift обеспечивает мощные функциональные средства управления пользователями, которые позволяют конкретизировать, что каждый пользователь может и не может делать. Эти функциональные средства предваряют управление ролевым доступом, которое теперь является стандартом в классической экосистеме Kubernetes.

Каждый пользователь имеет доступ к определенным проектам, которые являются не более чем пространствами имен Kubernetes с дополнительными аннотациями. Пользователи могут работать только с ресурсами, находящимися в проектах, к которым у пользователя есть доступ. Доступ к проекту предоставляется администратором кластера.

Знакомство с шаблонами приложений

Kubernetes позволяет разворачивать набор ресурсов с помощью одного манифеста JSON или YAML. OpenShift делает еще один шаг вперед, допуская параметризацию манифеста. Параметризуемый список в OpenShift называется *шаблоном*, это список объектов, определения которых могут содержать плейсхолдеры, которые будут заменены значениями параметров, когда вы будете их обрабатывать и затем создавать экземпляр шаблона (см. рис. 18.8).

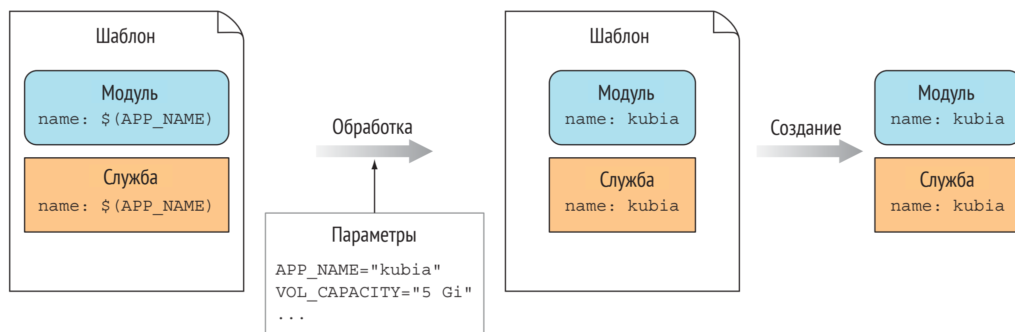


Рис. 18.8. Шаблоны OpenShift

Сам шаблон представляет собой файл JSON или YAML, содержащий список параметров, на которые имеются ссылки в ресурсах, определенных в том же файле JSON/YAML. Шаблон может храниться на сервере API, как и любой другой объект. Перед созданием экземпляра шаблона его необходимо обработать. Для обработки шаблона необходимо указать значения параметров шаблона, а затем OpenShift заменяет ссылки на параметры этими значениями. Результатом является обработанный шаблон, имеющий тот же вид, что список ресурсов Kubernetes, которые затем могут быть созданы с помощью одного запроса POST.

OpenShift предоставляет длинный список готовых шаблонов, которые позволяют пользователям быстро запускать сложные приложения, указав несколько аргументов (или вообще ни одного, если шаблон предоставляет для этих аргументов подходящие значения по умолчанию). Например, шаблон может позволить создать все ресурсы Kubernetes, необходимые для запуска приложения Java EE на прикладном сервере, который подключается к внутренней базе данных, также разворачиваемой в рамках того же шаблона. Все эти компоненты можно развернуть с помощью одной команды.

Сборка образов из исходного кода с помощью конфигураций сборок

Одной из лучших возможностей платформы OpenShift является возможность сборки и немедленного развертывания приложения в кластере OpenShift путем указания на репозиторий Git, содержащий исходный код приложения. Вам вообще не нужно создавать образ контейнера – OpenShift делает это за вас. Это происходит путем создания ресурса BuildConfig, который можно на-

строить для запуска сборок образов контейнеров сразу после внесения изменений в исходный репозиторий Git.

Хотя OpenShift не отслеживает сам репозиторий Git, специальный обработчик в репозитории может уведомлять OpenShift о новом коммите. Затем OpenShift извлекает изменения из репозитория Git и запускает процесс сборки. Механизм сборки под названием *Source To Image* (исходный код-в-образ) может определять тип приложения в репозитории Git и запускать для него соответствующую процедуру сборки. Например, если он обнаруживает файл pom.xml, который используется в проектах в формате Java Maven, то выполняет сборку Maven. Полученные артефакты упаковываются в соответствующий образ контейнера, а затем передаются во внутренний реестр контейнеров (предоставляемый платформой OpenShift). Оттуда их можно извлечь и немедленно запустить в кластере.

Создавая объект BuildConfig, разработчики могут указывать на репозиторий Git и не беспокоиться о создании образов контейнеров. Разработчикам почти не нужно ничего знать о контейнерах. Когда системные администраторы развертывают кластер OpenShift и предоставляют разработчикам к нему доступ, эти разработчики могут разрабатывать свой код, фиксировать и отправлять его в репозиторий Git точно так же, как они это делали до того, как мы начали упаковывать приложения в контейнеры. Затем OpenShift берет на себя создание, развертывание и управление приложениями из этого кода.

Автоматическое развертывание новых образов с помощью конфигураций развертывания

После создания нового образа контейнера его также можно автоматически развернуть в кластере. Это активируется путем создания объекта DeploymentConfig и указания его на ImageStream. Как следует из названия, ImageStream – это поток образов. Во время сборки образа он добавляется в поток образов ImageStream. Это позволяет конфигурации развертывания DeploymentConfig обнаруживать только что созданный образ и дает ей возможность предпринимать соответствующие действия и инициировать развертывание нового образа (см. рис. 18.9).

Объект DeploymentConfig почти идентичен объекту развертывания Deployment в Kubernetes, но он его предваряет. Как и объект Deployment, он имеет конфигурируемую стратегию перехода между развертываниями. Он содержит шаблон модуля, используемый для создания фактических модулей, но также позволяет конфигурировать обработчики пред- и постразвертывания. В отличие от развертывания в Kubernetes, вместо наборов реплик ReplicaSet он создает контроллеры репликации и предоставляет несколько дополнительных функциональных средств.

Предоставление доступа к службам извне с помощью маршрутов

На раннем этапе система Kubernetes не обеспечивала объекты Ingress. Для того чтобы предоставлять службы внешнему миру, приходилось использовать

службы типа NodePort или LoadBalancer. Но в то время OpenShift уже предоставлял более оптимальный вариант через ресурс маршрута Route. Объект Route подобен Ingress, но он предоставляет дополнительную конфигурацию, связанную с терминацией TLS и разделением трафика.

Подобно контроллеру Ingress, маршрут Route нуждается в маршрутизаторе Router, являющимся контроллером, который предоставляет балансировщика нагрузки или прокси. В отличие от Kubernetes, маршрутизатор Router в OpenShift доступен «из коробки».

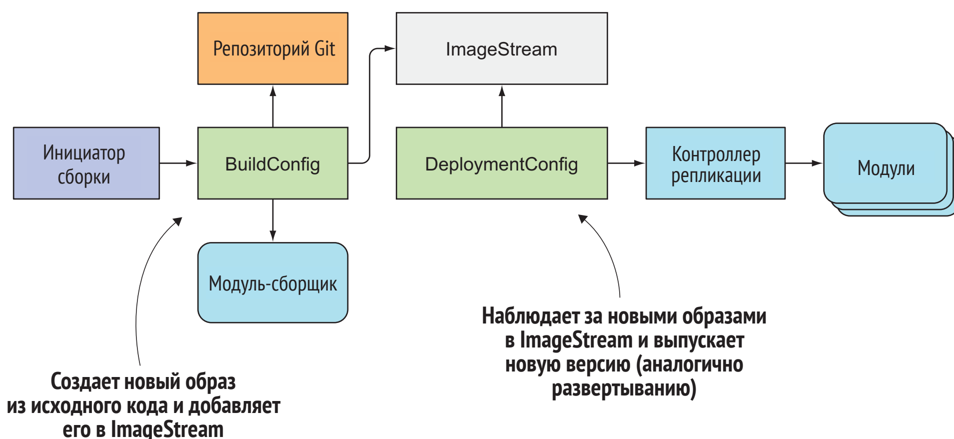


Рис. 18.9. Конфигурации сборок и конфигурации развертывания в OpenShift

Опробирование платформы OpenShift

Если вы заинтересованы в том, чтобы попробовать платформу OpenShift, то вы можете начать с помощью Minishift, который в OpenShift представляет собой эквивалент Minikube, или можете попробовать OpenShift Online Starter на https://manage.openshift.com, представляющий собой бесплатное мультитенантное размещенное на своих ресурсах решение, с которого вы можете начать работу с платформой OpenShift.

18.3.2 Deis Workflow и Helm

Компания под названием Deis, которая недавно была приобретена Microsoft, также предоставляет PaaS-систему под названием Workflow, которая тоже надстроена поверх Kubernetes. Помимо Workflow, в этой компании также разработали инструмент под названием Helm, который набирает обороты в сообществе Kubernetes как стандартный способ развертывания существующих приложений в Kubernetes. Мы кратко рассмотрим оба инструмента.

Знакомство с Deis Workflow

Deis Workflow можно развернуть в любом существующем кластере Kubernetes (в отличие от платформы OpenShift, которая является полным кластером с модифицированным сервером API и другими компонентами Kubernetes). При

запуске Workflow создается набор служб и контроллеров репликации, которые затем предоставляют разработчикам простую и удобную для них среду.

Развертывание новых версий приложения инициируется отправкой изменений с помощью команды `git push deis master`, предоставив системе Workflow позаботиться об остальном. Подобно платформе OpenShift, система Workflow также предоставляет механизм исходный код-в-образ, раскрутки и откатов приложений, периферийной маршрутизации, а также агрегацию журналов, метрики и предупреждения, которые недоступны в ядре Kubernetes.

Для запуска Workflow в кластере Kubernetes в первую очередь необходимо установить Deis Workflow, инструмент командной строки Helm и затем установить Workflow в кластере. Мы не будем вдаваться в подробности того, как это сделать, но если вы хотите узнать больше, посетите веб-сайт по адресу https://deis.com/workflow. Здесь же мы рассмотрим инструмент Helm, который можно использовать без Workflow и который приобрел популярность в сообществе Kubernetes.

Развертывание ресурсов через Helm

Helm – это менеджер пакетов для Kubernetes (подобно менеджерам пакетов ОС, таким как `yum` или `apt` в Linux или `homebrew` в MacOS).

Helm состоит из двух компонентов:

- инструмента командной строки `helm` (CLI-клиент);
- Tiller – серверного компонента, работающего как модуль в кластере Kubernetes.

Эти два компонента используются для развертывания пакетов приложений и управления ими в кластере Kubernetes. Пакеты приложений Helm называются схемами Chart. Они образуют конфигурацию Config, которая содержит сведения о конфигурации и объединяется в схему для создания релиза Release, представляющего собой работающий экземпляр приложения (комбинацию схемы и конфигурации). Вы развертываете релизы и управляете ими с помощью инструмента командной строки `helm`, который обращается к серверу Tiller, то есть к компоненту, создающему все необходимые ресурсы Kubernetes, определенные в схеме, как показано на рис. 18.10.

Вы можете создавать схемы самостоятельно и сохранять их на локальном диске, либо вы можете использовать любую существующую схему из имеющихся в растущем списке схем Helm, поддерживаемых сообществом по адресу https://github.com/kubernetes/charts. В данный список входят схемы для таких приложений, как PostgreSQL, MySQL, MariaDB, Magento, Memcached, MongoDB, OpenVPN, PHPBB, RabbitMQ, Redis, WordPress и др.

Подобно тому, как вы не делаете ручную сборку и инсталляцию в вашу систему Linux приложений, разработанных другими людьми, вы, вероятно, не захотите создавать и управлять своими собственными манифестами Kubernetes для таких приложений, не правда ли? Вот почему вам потребуется использовать Helm и схемы, доступные в репозитории GitHub, о котором я упоминал.

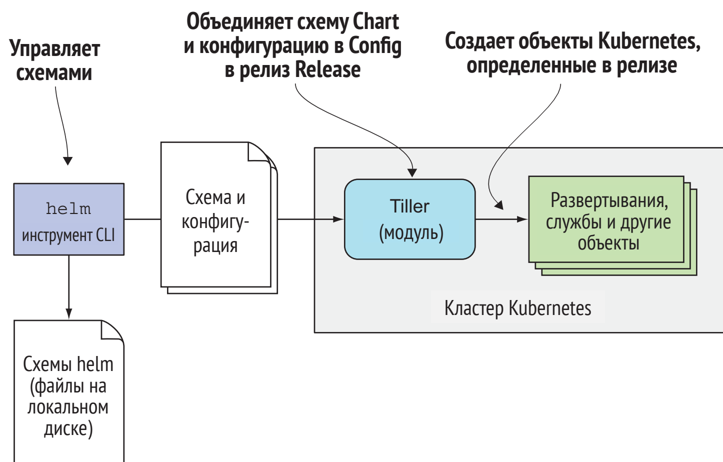


Рис. 18.10. Общий вид менеджера пакетов Helm

Если вы хотите запустить базу данных PostgreSQL или MySQL в кластере Kubernetes, не начинайте писать для них манифесты. Вместо этого проверьте, может, кто-то уже прошел через все неприятности и подготовил для нее схему Helm.

После того как кто-то подготавливает схему конкретного приложения и добавляет ее в репозиторий Github схем Helm, установка всего приложения занимает одну однострочную команду. Например, чтобы запустить СУБД MySQL в кластере Kubernetes, от вас требуется только клонировать схемы с репозиторием Git на локальный компьютер и выполнить следующую ниже команду (при условии что у вас есть инструмент командной строки `helm` и сервер Tiller работает в вашем кластере):

```
$ helm install --name my-database stable/mysql
```

Эта команда создаст все необходимые развертывания, службы, секреты и заявки PersistentVolumeClaim, необходимые для выполнения MySQL в вашем кластере. Вам не нужно беспокоиться о том, какие компоненты вам нужны и как их настроить для правильной работы MySQL. Я уверен, вы согласитесь, что это потрясающе.

СОВЕТ. Одна из самых интересных схем, доступных в репозитории, – это схема OpenVPN, которая запускает сервер OpenVPN в кластере Kubernetes и позволяет вам входить в сеть модулей через VPN и обращаться к службам так, как если бы ваша локальная машина была модулем в кластере. Это пригодится, когда вы разрабатываете приложения и запускаете их локально.

Выше было показано несколько примеров того, как Kubernetes может быть расширен и как компании, такие как Red Hat и Deis (теперь Microsoft), работают над его расширением. Теперь идите и оседлайте свою волну на Kubernetes!

18.4 Резюме

Эта заключительная глава показала, как вы можете выйти за рамки существующих функциональных возможностей, которые обеспечивает Kubernetes, и как некоторые компании, такие как Dies и Red Hat, это сделали. Вы познакомились с тем, как:

- можно зарегистрировать свои собственные ресурсы на сервере API, создав объект CustomResourceDefinition;
- могут храниться, извлекаться, обновляться и удаляться экземпляры пользовательских объектов без необходимости изменения кода сервера API;
- можно реализовать пользовательский контроллер, который будет оживлять эти объекты;
- можно расширить экосистему Kubernetes с помощью пользовательских серверов API через агрегацию API;
- каталог служб Kubernetes позволяет саморезервировать внешние службы и предоставлять к ним доступ модулям, работающим в кластере Kubernetes;
- платформы как службы (PaaS), построенные поверх Kubernetes, упрощают создание контейнерных приложений внутри одного и того же кластера Kubernetes, который затем их запускает;
- менеджер пакетов под названием Helm позволяет развертывать существующие приложения без необходимости создавать для них манифесты ресурсов.

Благодарю, что нашли время прочитать эту длинную книгу. Надеюсь, вы узнали из нее столько же, сколько я узнал во время ее написания.

Приложение A

Использование kubectl с несколькими кластерами

A.1 Переключение между Minikube и Google Kubernetes Engine

Примеры в этой книге могут быть выполнены в кластере, созданном либо с помощью Minikube, либо с помощью Kubernetes Kubernetes Engine (GKE). Если вы планируете использовать оба, то вам нужно знать, как переключаться между ними. Подробное описание использования инструмента командной строки `kubectl` с несколькими кластерами описано в следующем далее разделе. Здесь же мы рассмотрим, как переключаться между Minikube и GKE.

Переключение на Minikube

К счастью, всякий раз, когда вы запускаете кластер Minikube с помощью команды `minikube start`, она также перенастраивает инструмент `kubectl`, чтобы его можно было с ним использовать:

```
$ minikube start
Starting local Kubernetes cluster...
...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

Minikube настраивает kubectl всякий раз, когда вы запускаете кластер

После переключения с Minikube на GKE можно переключиться обратно, остановив Minikube и запустив его снова. Затем `kubectl` будет повторно настроен, чтобы снова использовать кластер Minikube.

Переключение на GKE

Для переключения на использование кластера GKE можно использовать следующую ниже команду:

```
$ gcloud container clusters get-credentials my-gke-cluster
```

Эта команда позволит настроить `kubectl` для использования кластера GKE под названием `my-gke-cluster`.

Дальнейшие шаги

Этих двух методов должно быть достаточно для быстрого старта, но для того, чтобы понять полную картину использования инструмента `kubectl` с многочисленными кластерами, изучите следующий ниже раздел.

А.2 Использование инструмента kubectl со множеством кластеров или пространств имен

Если вам нужно переключаться между различными кластерами Kubernetes или вы хотите работать в пространстве имен, отличном от пространства имен, установленном по умолчанию, и не хотите указывать параметр `--namespace` при каждом запуске инструмента `kubectl`, то ниже показано, что нужно сделать.

А.2.1 Настройка расположения файла kubeconfig

Конфигурация, которая используется инструментом `kubectl`, обычно хранится в файле ~/.kube/config. Если она хранится где-то еще, то переменная среды `KUBECONFIG` должна указывать на ее расположение.

> **ПРИМЕЧАНИЕ.** Вы можете использовать несколько файлов конфигурации, и `kubectl` будет все их использовать сразу. Для этого укажите их все в переменной среды `KUBECONFIG` (разделив их двоеточием).

А.2.2 Содержимое файла kubeconfig

Пример файла конфигурации показан в следующем ниже листинге.

Листинг А.1. Пример файла kubeconfig

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/luksa/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
    namespace: default
  name: minikube
current-context: minikube
kind: Config
```

Содержит информацию о кластере Kubernetes (clusters)

Определяет контекст kubectl (contexts)

Текущий контекст, используемый kubectl (current-context)

```
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/luksa/.minikube/apiserver.crt
   client-key: /home/luksa/.minikube/apiserver.key
```

Содержит учетные данные пользователя

Файл kubeconfig состоит из четырех секций:

- список кластеров;
- список пользователей;
- список контекстов;
- имя текущего контекста.

У каждого кластера, пользователя и контекста есть имя. Имя используется для ссылки на контекст, пользователя или кластер.

Кластеры

Секция `cluster` представляет кластер Kubernetes и содержит URL-адрес сервера API, файл центра сертификации (CA) и, возможно, несколько других параметров конфигурации, связанных со взаимодействием с сервером API. Сертификат CA может храниться в отдельном файле, и файл kubeconfig может содержать на него ссылку, либо он может быть включен в файл непосредственно в поле `certificate-authority-data`.

Пользователи

В каждой секции `user` определяются учетные данные для использования при обращении к серверу API. Это может быть пара из имени пользователя и пароля, токен аутентификации или клиентский ключ и сертификат. Сертификат и ключ могут быть включены в файл kubeconfig (через свойства `clientcertificate-data` и `client-key-data`) или сохранены в отдельных файлах, ссылки на которые указываются в файле конфигурации, как показано в листинге A.1.

Контексты

Секция `context` связывает кластер, пользователя и пространство имен, которое будет использоваться по умолчанию, которые инструмент `kubectl` должен использовать при выполнении команд. Несколько контекстов могут указывать на одного и того же пользователя или кластер.

Текущий контекст

Хотя в файле kubeconfig может быть определено несколько контекстов, в любой момент времени только один из них является текущим контекстом. Позже мы увидим, как можно менять текущий контекст.

A.2.3 Вывод списка, добавление и изменение записей в файле kubeconfig

Данный файл можно редактировать вручную, добавляя, изменяя и удаляя кластеры, пользователей или контексты, но это также можно сделать с помощью одной из команд `kubectl config`.

Добавление или изменение кластера

Для того чтобы добавить еще один кластер, примените команду `kubectl config set-cluster`:

```
$ kubectl config set-cluster my-other-cluster
➥  --server=https://k8s.example.com:6443
➥  --certificate-authority=path/to/the/cafile
```

Эта команда добавит кластер под названием `my-other-cluster` с сервером API, расположенным по адресу https://k8s.example.com:6443. Для того чтобы просмотреть дополнительные параметры, которые вы можете передать в команду, выполните команду `kubectl config set-cluster`, чтобы распечатать примеры использования.

Если кластер с таким именем уже существует, то команда `set-cluster` перезапишет его конфигурацию.

Добавление или изменение учетных данных пользователя

Добавление и изменение пользователей аналогично добавлению или изменению кластера. Для того чтобы добавить пользователя, который аутентифицируется на сервере API, используя имя пользователя и пароль, выполните следующую ниже команду:

```
$ kubectl config set-credentials foo --username=foo --password=pass
```

Для того чтобы использовать аутентификацию на основе токена, выполните следующие ниже действия:

```
$ kubectl config set-credentials foo --token=mysecrettokenXFDJIQ1234
```

В обоих примерах учетные данные пользователя хранятся под именем `foo`. Если для аутентификации в разных кластерах используются одни и те же учетные данные, можно определить одного пользователя и использовать его в обоих кластерах.

Связывание кластеров и учетных данных пользователей

Контекст определяет, какого пользователя использовать с каким кластером, но также может определять пространство имен, которое должен использовать `kubectl`, если вы не указываете пространство имен явно с помощью параметра `--namespace` или `-n`.

Следующая ниже команда используется для создания нового контекста, связывающего кластер и созданного пользователя:

```
$ kubectl config set-context some-context --cluster=my-other-cluster
➥   --user=foo --namespace=bar
```

Она создает контекст под названием `some-context`, который применяет кластер `my-other-cluster` и учетные данные пользователя `foo`. Пространство имен, которое будет использоваться по умолчанию, в этом контексте имеет значение `bar`.

К примеру, вы также можете использовать ту же команду для изменения пространства имен текущего контекста. Имя текущего контекста можно получить следующим образом:

```
$ kubectl config current-context
minikube
```

Затем можно изменить пространство имен:

```
$ kubectl config set-context minikube --namespace=another-namespace
```

Выполнение этой простой команды один раз намного удобнее, по сравнению с включением параметра `--namespace` при каждом выполнении `kubectl`.

> **СОВЕТ.** Для того чтобы легко переключаться между пространствами имен, определите такой псевдоним: `aalias kcd='kubectl config set-context $(kubectl config current-context) --namespace'`, после чего можно переключаться между пространствами имен с помощью `kcd somenamespace`.

А.2.4 Использование инструмента kubectl с разными кластерами, пользователями и контекстами

При выполнении команд `kubectl` используются кластер, пользователь и пространство имен, определенные в текущем контексте файла kubeconfig, но их можно переопределить с помощью следующих далее параметров командной строки:

- `--user` для использования другого пользователя из файла kubeconfig;
- `--username` и `--password` для использования другого имени пользователя и/или пароля (их не нужно указывать в конфигурационном файле). При использовании других типов аутентификации можно использовать `--client-key` и `--client-certificate` или `-token`;
- `--cluster` для использования другого кластера (должен быть определен в файле конфигурации);
- `--server` для указания URL-адреса другого сервера (которого нет в конфигурационном файле);
- `--namespace` для использования другого пространства имен.

А.2.5 Переключение между контекстами

Вместо изменения текущего контекста, как в одном из предыдущих примеров, для создания дополнительного контекста и переключения между контекстами можно также использовать команду `set-context`. Это удобно при работе с несколькими кластерами (для их создания используйте `set-cluster`).

После того как вы настроили несколько контекстов, переключение между ними выполняется тривиальным образом:

```
$ kubectl config use-context my-other-context
```

Эта команда переключает текущий контекст на `my-other-context`.

А.2.6 Перечисление контекстов и кластеров

Чтобы вывести список всех контекстов, определенных в файле kubeconfig, выполните следующую ниже команду:

```
$ kubectl config get-contexts
CURRENT   NAME          CLUSTER       AUTHINFO            NAMESPACE
*         minikube      minikube      minikube            default
          rpi-cluster   rpi-cluster   admin/rpi-cluster
          rpi-foo       rpi-cluster   admin/rpi-cluster   foo
```

Как видите, я использую три разных контекста. Контексты `rpi-cluster` и `rpi-foo` используют один и тот же кластер и учетные данные, но по умолчанию применяют разные пространства имен. Список кластеров похож:

```
$ kubectl config get-clusters
NAME
rpi-cluster
minikube
```

Учетные данные не могут быть перечислены по соображениям безопасности.

А.2.7 Удаление контекстов и кластеров

Чтобы очистить список контекстов или кластеров, записи из файла kubeconfig можно удалить вручную либо применить следующие ниже две команды:

```
$ kubectl config delete-context my-unused-context
```

и

```
$ kubectl config delete-cluster my-old-cluster
```

Приложение B

Настройка многоузлового кластера с помощью kubeadm

В этом приложении показано, как установить кластер Kubernetes с несколькими узлами. Вы будете запускать узлы внутри виртуальных машин посредством программы виртуализации VirtualBox, но вы также можете использовать другой инструмент виртуализации или машины без операционной системы. Для того чтобы настроить ведущий и рабочие узлы, вы будете использовать инструмент `kubeadm`.

B.1 Настройка ОС и необходимых пакетов

Прежде всего вам нужно скачать и установить программу VirtualBox, если у вас ее еще нет. Вы можете скачать ее с https://www.virtualbox.org/wiki/Downloads. После того как вы ее установили, следует скачать дистрибутив Linux – минимальный ISO-образ CentOS 7 с www.centos.org/download. Вы также можете использовать другой дистрибутив, но убедитесь, что он поддерживается, обратившись на сайт http://kubernetes.io.

B.1.1 Создание виртуальной машины

Затем вы создадите виртуальную машину для ведущего узла Kubernetes. Начните с нажатия на значок в верхнем левом углу. Затем в качестве имени введите «k8s-master» и выберите Linux в качестве типа и Red Hat (64-bit) в качестве версии, как показано на рис. B.1.

После нажатия кнопки **Next** (**Далее**) можно задать размер памяти виртуальной машины и настроить жесткий диск. Если у вас достаточно памяти, выберите, по крайней мере, 2 Гб (имейте в виду, что вы будете запускать три такие виртуальные машины). При создании жесткого диска оставьте параметры по умолчанию. Вот какими они были в моем случае:

- тип файла жесткого диска: VDI (образ диска VirtualBox);
- хранилище на физическом жестком диске: динамически выделенное;
- расположение и размер файла: k8s-master, размер 8 Гб.

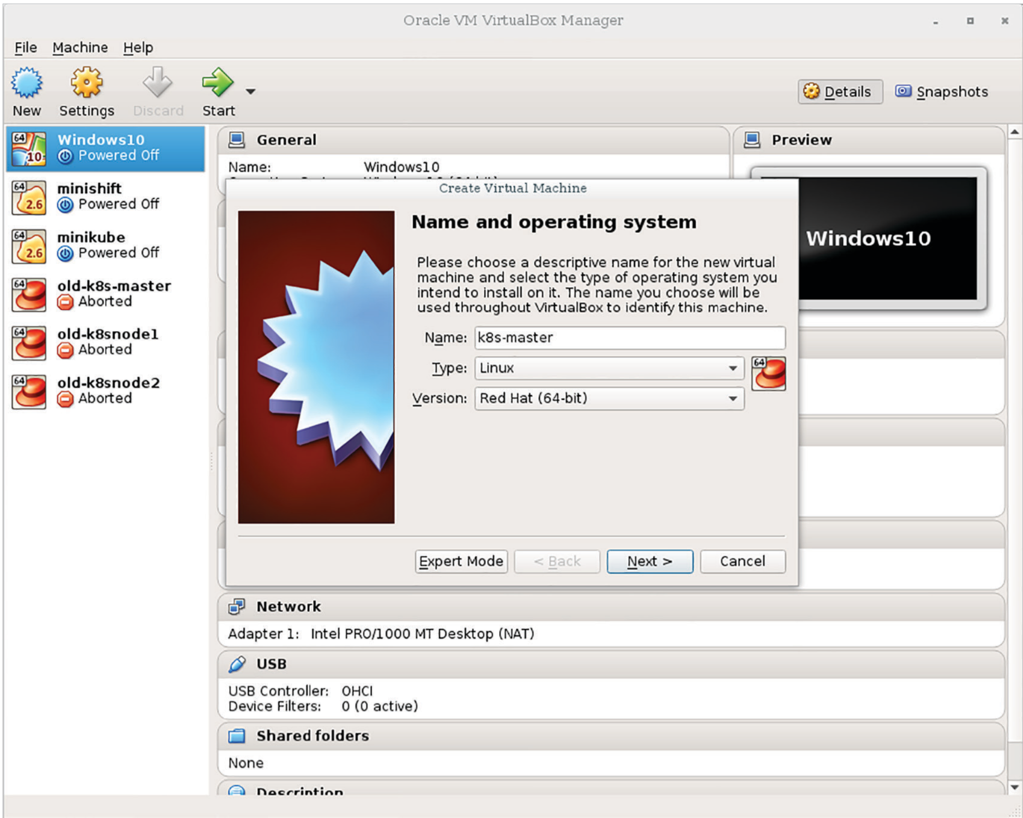


Рис. B.1. Создание виртуальной машины в VirtualBox

B.1.2 Настройка сетевого адаптера для виртуальной машины

После создания виртуальной машины необходимо настроить ее сетевой адаптер, так как по умолчанию невозможно правильно запустить несколько узлов. Вы настроите адаптер таким образом, чтобы он использовал режим мостового адаптера. Это позволит подключить виртуальные машины к той же сети, в которой находится хост-компьютер. Каждая виртуальная машина получит свой собственный IP-адрес, так же, как если бы это была физическая машина, подключенная к тому же коммутатору, к которому подключен ваш хост-компьютер. Другие параметры гораздо сложнее, так как обычно для их настройки требуется два сетевых адаптера.

Для того чтобы настроить сетевой адаптер, убедитесь, что виртуальная машина выбрана в главном окне Virtual-Box, а затем щелкните по значку **Settings** (**Параметры**) (рядом со значком **New** (**Новый**), по которому вы щелкнули ранее).

Появится окно, подобное показанному на рис. B.2. На левой стороне выберите **Network** (**Сеть**), а затем на главной панели справа выберите **Attached**

to (**Подключено к**): **Bridged Adapter** (**Мостовой адаптер**), как показано на рисунке. В раскрывающемся меню **Name** (**Имя**) выберите адаптер вашей хост-машины, который используется для подключения компьютера к сети.

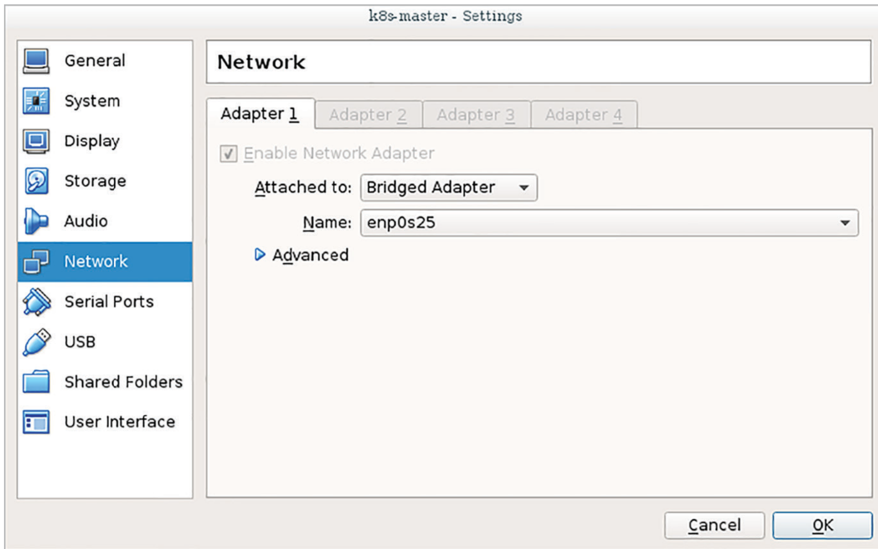


Рис. B.2. Настройка сетевого адаптера для виртуальной машины

B.1.3 Инсталляция операционной системы

Теперь можно запустить виртуальную машину и инсталлировать операционную систему. Убедитесь, что виртуальная машина по-прежнему выбрана в списке, и щелкните по значку **Start** (**Пуск**) в верхней части главного окна VirtualBox.

Выбор загрузочного диска

Перед запуском виртуальной машины программа VirtualBox запросит, какой загрузочный диск использовать. Щелкните по значку рядом с раскрывающимся списком (показано на рис. B.3), а затем найдите и выберите ISO-образ CentOS, скачанный ранее. Потом нажмите кнопку **Start** (**Пуск**), чтобы выполнить начальную загрузку виртуальной машины.

Запуск инсталляции

При запуске виртуальной машины появится текстовое меню. С помощью клавиши перемещения курсора вверх выберите опцию **Установить CentOS Linux 7** и нажмите кнопку **Enter**.

Настройка параметров инсталляции

Через несколько мгновений появится графический экран приветствия CentOS Linux 7, позволяющий выбрать язык, который вы хотите использо-

вать. Я советую оставить английский язык. Нажмите кнопку **Continue** (**Продолжить**), чтобы перейти к главному экрану настройки, как показано на рис. В.4.

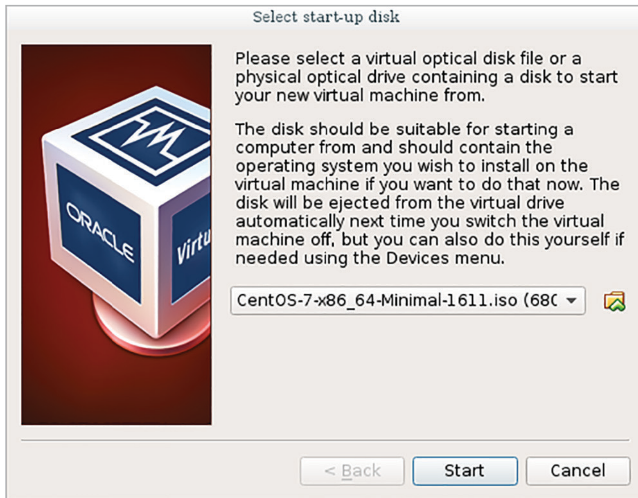


Рис. В.3. Выбор инсталляционного ISO-образа

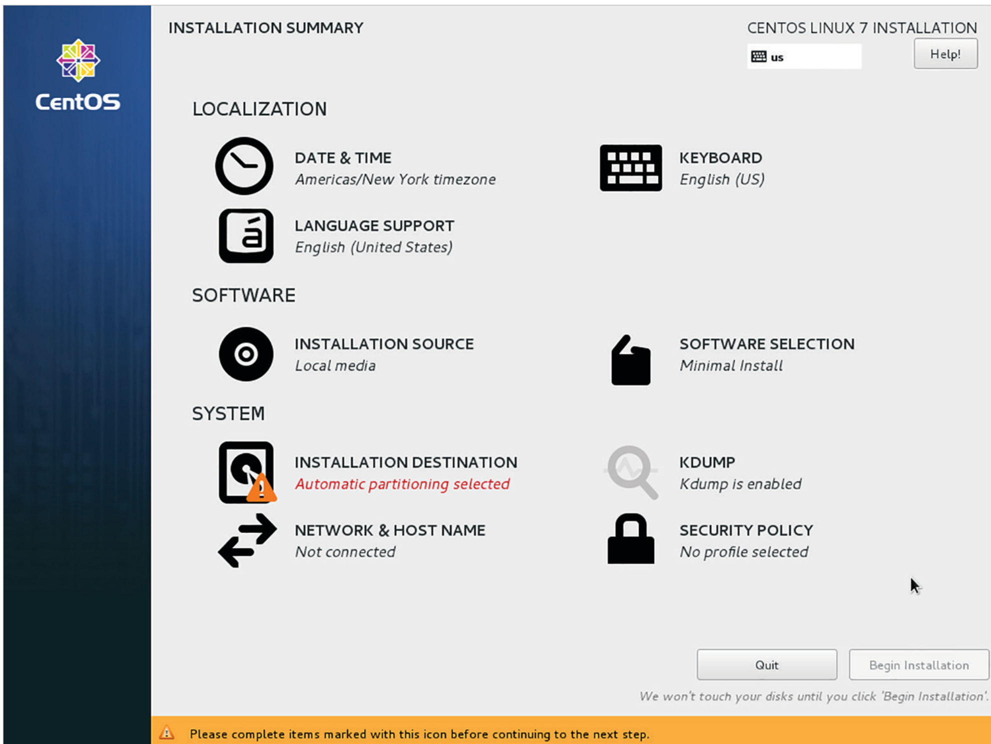


Рис. В.4. Главный экран настройки

> **СОВЕТ.** Когда вы щелкаете мышью в окне виртуальной машины, клавиатура и мышь будут захвачены виртуальной машиной. Для того чтобы их освободить, нажмите клавишу, показанную в правом нижнем углу окна VirtualBox, в котором запущена виртуальная машина. Обычно это правая клавиша управления в Windows и Linux или левая клавиша управления в MacOS.

Сначала нажмите кнопку **Installation Destination** (**Место установки**), а затем на появившемся экране сразу же нажмите кнопку **Done** (**Готово**) (вам не нужно нажимать в другом месте).

Затем нажмите на **Network & Host Name** (**Сеть и хостнейм**). На следующем экране сначала включите сетевой адаптер, нажав переключатель **ON/OFF** в правом верхнем углу. Затем введите имя в поле внизу слева, как показано на рис. B.5. Сейчас вы настраиваете ведущий узел, поэтому установите сетевое имя в master.k8s. Нажмите кнопку **Apply** (**Применить**) рядом с текстовым полем, чтобы подтвердить новый хостнейм.

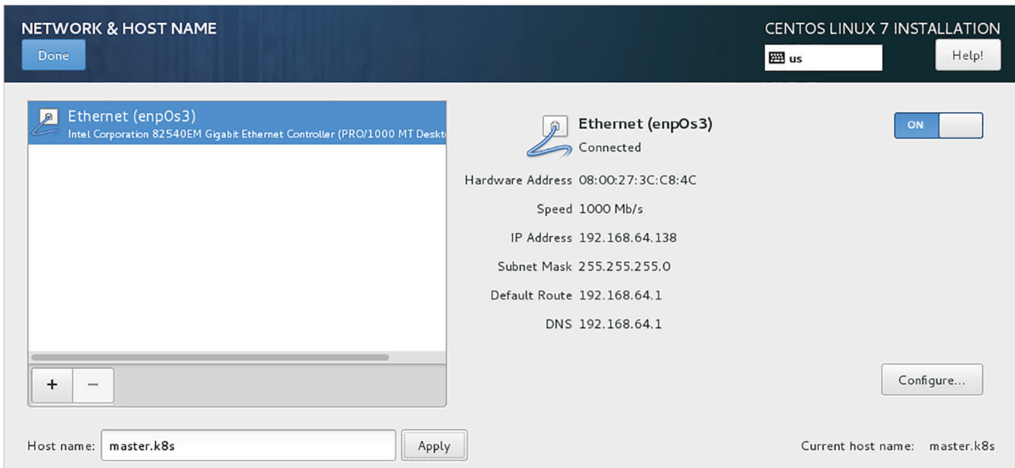


Рис. B.5. Настройка хостнейма и конфигурирование сетевого адаптера

Для того чтобы вернуться на главный экран настройки, нажмите кнопку **Done** (**Готово**) в левом верхнем углу.

Также нужно установить правильный часовой пояс. Щелкните **Date & Time** (**Дата и время**), а затем на открывшемся экране выберите регион и город или щелкните свое местоположение на карте. Вернитесь на главный экран, нажав кнопку **Done** (**Готово**) в левом верхнем углу.

Запуск инсталляции

Для того чтобы начать инсталляцию, нажмите кнопку **Begin Installation** (**Начать установку**) в правом нижнем углу. Появится экран, подобный показанному на рис. B.6. Во время инсталляции ОС, если вы хотите, установите пароль root и создайте учетную запись пользователя. После завершения инсталляции нажмите кнопку перезагрузки в правом нижнем углу.

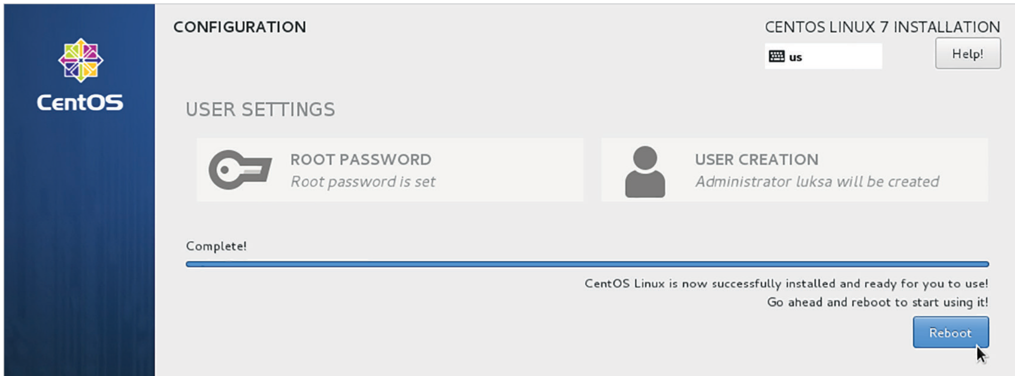


Рис. В.6. Установка пароля root во время установки ОС и последующая перезагрузка

В.1.4 Установка Docker и Kubernetes

Войдите в систему как root. Сначала необходимо отключить две функции безопасности: SELinux и брендмауэр.

Отключение системы контроля SELinux

Для того чтобы отключить систему контроля SELinux, выполните следующую ниже команду:

```
# setenforce 0
```

Но она отключает ее только временно (до следующей перезагрузки). Для того чтобы отключить ее навсегда, отредактируйте файл /etc/selinux/config и измените строку `SELINUX=enforcing` на `SELINUX=permissive`.

Отключение брендмауэра

Для того чтобы не столкнуться с проблемами, связанными с брендмауэром, вы также отключите брендмауэр. Выполните следующую ниже команду:

```
# systemctl disable firewalld && systemctl stop firewalld
Removed symlink /etc/systemd/system/dbus-org.fedoraproject.FirewallD1...
Removed symlink /etc/systemd/system/basic.target.wants/firewalld.service.
```

Добавление репозитория Kubernetes yum

Для того чтобы пакеты Kubernetes RPM стали доступными менеджеру пакетов yum, вы добавите файл kubernetes.repo в каталог /etc/yum.repos.d/, как показано в следующем ниже листинге.

Листинг В.1. Добавление репозитория Kubernetes RPM

```
# cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
```

```
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

ПРИМЕЧАНИЕ. Если вы копируете и вставляете, проверьте, чтобы не было пробелов после EOF.

Инсталляция Docker, Kubelet, kubeadm, kubectl и Kubernetes-CNI

Теперь вы готовы установить все пакеты, которые вам нужны:

```
# yum install -y docker kubelet kubeadm kubectl kubernetes-cni
```

Как видите, вы устанавливаете довольно много пакетов. Вот они:

- `docker` – контейнерная среда выполнения;
- `kubelet` – агент узла Kubernetes, который будет для вас все запускать;
- `kubeadm` – инструмент для развертывания многоузловых кластеров Kubernetes;
- `kubectl` – инструмент командной строки для взаимодействия с Kubernetes;
- `kubernetes-cni` – интерфейс контейнерного сетевого взаимодействия Kubernetes.

После их установки необходимо вручную активировать `docker` и службы `kubelet`:

```
# systemctl enable docker && systemctl start docker
# systemctl enable kubelet && systemctl start kubelet
```

Активация net.bridge.bridge-nf-call-iptables в ядре

Я заметил, что что-то деактивирует ключевой параметр `bridge-nf-call-iptables`, который необходим для правильной работы служб Kubernetes. Для того чтобы устранить данную проблему, необходимо выполнить следующие две команды:

```
# sysctl -w net.bridge.bridge-nf-call-iptables=1
# echo "net.bridge.bridge-nf-call-iptables=1" > /etc/sysctl.d/k8s.conf
```

Отключение swap

Kubelet не будет работать, если включен swap, поэтому вы отключите его с помощью следующей ниже команды:

```
# swapoff -a && sed -i '/ swap / s/^/#/' /etc/fstab
```

В.1.5 Клонирование виртуальной машины

Все, что вы сделали до этого момента, должно быть сделано на каждой машине, которую вы планируете использовать в кластере. Если вы делаете это на машине без операционной системы (на голом железе), то вам нужно повторить процесс, описанный в предыдущем разделе, по крайней мере еще два раза – для каждого рабочего узла. Если вы создаете кластер с помощью виртуальных машин, то сейчас самое время клонировать виртуальную машину, чтобы в итоге получить три разные виртуальные машины.

Завершение работы виртуальной машины

Для того чтобы клонировать машину в VirtualBox, сначала завершите работу виртуальной машины, выполнив команду shutdown:

```
# shutdown now
```

Клонирование виртуальной машины

Теперь щелкните правой кнопкой мыши на виртуальной машине в пользовательском интерфейсе VirtualBox и выберите **Clone** (**Клонировать**). Введите имя новой машины, как показано на рис. В.7 (например, k8s-node1 для первого клона или k8s-node2 для второго). Убедитесь, что установлен флажок **Reinitialize the MAC address of all network cards** (**Повторно инициализировать MAC-адрес всех сетевых карт**), чтобы каждая виртуальная машина использовала разные MAC-адреса (поскольку они будут расположены в одной сети).

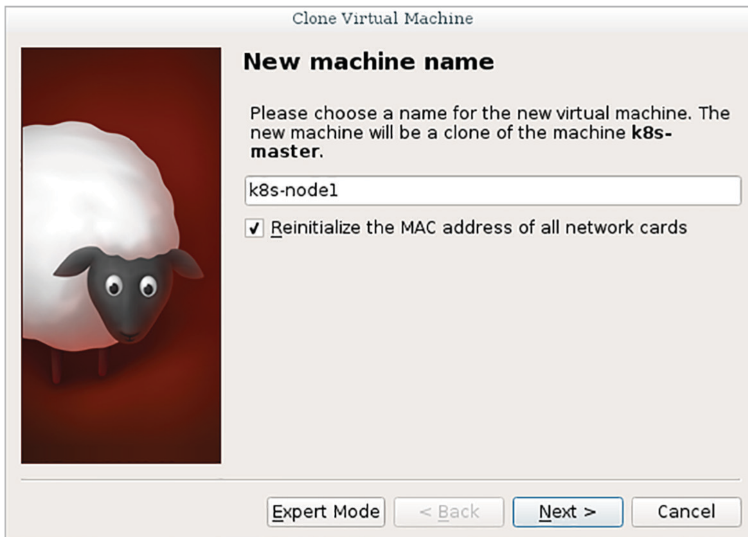


Рис. В.7. Клонирование ведущей виртуальной машины

Нажмите кнопку **Next** (**Далее**), а затем, перед тем как снова нажать кнопку **Next** (**Далее**), убедитесь, что выбран параметр **Full clone** (**Полное клониро-

вание). Затем на следующем экране нажмите кнопку **Clone** (**Клонировать**); оставьте выбранным параметр **Current machine state** (**Текущее состояние машины**).

Повторите данный процесс для виртуальной машины для второго узла, а затем запустите все три виртуальные машины, выбрав все три и щелкнув значок **Start** (**Пуск**).

Изменение хостнейма имени на клонированных виртуальных машинах

Поскольку вы создали два клона из ведущей виртуальной машины, все три виртуальные машины имеют одинаковый хостнейм. Поэтому необходимо изменить хостнейм двух клонов. Для этого войдите в каждый из двух узлов (как root) и выполните следующую ниже команду:

```
# hostnamectl --static set-hostname node1.k8s
```

ПРИМЕЧАНИЕ. На втором узле обязательно назначьте хостнейм `node2.k8s`.

Настройка разрешения имен для всех трех хостов

Необходимо убедиться, что все три узла разрешимы (resolvable) либо путем добавления записей на DNS-сервер, либо путем редактирования файла /etc/hosts на всех из них. Например, вы должны добавить следующие три строки в файл hosts (заменить IP-адреса на адреса виртуальных машин), как показано в следующем ниже листинге.

Листинг В.2. Записи для добавления в /etc/hosts на каждом узле кластера

```
192.168.64.138 master.k8s
192.168.64.139 node1.k8s
192.168.64.140 node2.k8s
```

Вы можете получить IP-адрес каждого узла, войдя в узел как root, выполнив `ip addr` и найдя IP-адрес, связанный с сетевым адаптером `enp0s3`, как показано в следующем ниже листинге.

Листинг В.3. Поиск IP-адреса каждого узла

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
```

```
    UP qlen 1000
    link/ether 08:00:27:db:c3:a4 brd ff:ff:ff:ff:ff:ff
    inet 192.168.64.138/24 brd 192.168.64.255 scope global dynamic enp0s3
       valid_lft 59414sec preferred_lft 59414sec
    inet6 fe80::77a9:5ad6:2597:2e1b/64 scope link
       valid_lft forever preferred_lft forever
```

Результат команды в приведенном выше листинге показывает, что IP-адресом машины является `192.168.64.138`. Для того чтобы получить все их IP-адреса, вам нужно будет выполнить эту команду на каждом из ваших узлов.

B.2 Конфигурирование ведущего узла с помощью kubeadm

Теперь вы наконец готовы настроить плоскость управления Kubernetes на ведущем узле.

Выполнение kubeadm init для инициализации ведущего узла

Благодаря удивительному инструменту kubeadm, для того чтобы инициализировать ведущий узел, от вас требуется только выполнить одну-единственную команду, как показано в следующем ниже листинге.

Листинг B.4. Инициализация мастер-узла с помощью команды kubeadm init

```
# kubeadm init
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production
     clusters.
[init] Using Kubernetes version: v.1.8.4
...
You should now deploy a pod network to the cluster.
Run «kubectl apply -f [podnetwork].yaml» with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/
You can now join any number of machines by running the following on each node
     as root:
kubeadm join --token eb3877.3585d0423978c549 192.168.64.138:6443
     --discovery-token-ca-cert-hash
     sha256:037d2c5505294af196048a17f184a79411c7b1eac48aaa0ad137075be3d7a847
```

> **ПРИМЕЧАНИЕ.** Запишите команду, показанную в последней строке результата kubeadm init. Вам она понадобится позже.

Kubeadm развернул все необходимые компоненты плоскости управления, включая хранилище etcd, сервер API, планировщик и менеджер контроллеров. Он также развернул kube-proxy, сделав службы Kubernetes доступными из ведущего узла.

В.2.1 Как kubeadm запускает компоненты

Все эти компоненты работают как контейнеры. Для подтверждения вы можете применить команду `docker ps`. Однако `kubeadm` не использует платформу Docker напрямую для их запуска. Он развертывает их дескрипторы YAML в каталоге /etc/kubernetes/manifests. Этот каталог отслеживается агентом Kubelet, который затем запускает эти компоненты через платформу Docker. Компоненты работают как модули. Вы можете увидеть их с помощью команды `kubectl get`. Но сначала необходимо настроить инструмент командной строки `kubectl`.

Запуск kubectl на ведущем узле

Вы установили `kubectl` вместе с `docker`, `kubeadm` и другими пакетами на одном из начальных этапов. Но `kubectl` нельзя использовать для связи с кластером без предварительной настройки с помощью файла kubeconfig.

К счастью, необходимая его конфигурация хранится в файле /etc/kubernetes/admin.conf. Вам нужно лишь сделать так, чтобы `kubectl` его использовал, установив для этого переменную среды `KUBECONFIG`, как описано в приложении А:

```
# export KUBECONFIG=/etc/kubernetes/admin.conf
```

Вывод списка модулей

Для тестирования `kubectl` вы можете вывести список модулей плоскости управления (они находятся в пространстве имен `kube-system`), как показано в следующем ниже листинге.

Листинг В.5. Системные модули в пространстве имен kube-system

```
# kubectl get po -n kube-system
NAME                               READY  STATUS   RESTARTS  AGE
etcd-master.k8s                    1/1    Running  0         21m
kube-apiserver-master.k8s          1/1    Running  0         22m
kube-controller-manager-master.k8s 1/1    Running  0         21m
kube-dns-3913472980-cn6kz          0/3    Pending  0         22m
kube-proxy-qb709                   1/1    Running  0         22m
kube-scheduler-master.k8s          1/1    Running  0         21m
```

Вывод списка узлов

Вы закончили с настройкой ведущего узла, но вам все еще нужно настроить узлы. Хотя агент Kubelet уже установлен на обоих рабочих узлах (каждый узел устанавливается отдельно или клонируется после установки всех необходимых пакетов), они еще не являются частью кластера Kubernetes. Это можно увидеть, выведя список узлов с помощью `kubectl`:

```
# kubectl get node
NAME       STATUS    ROLES   AGE  VERSION
master.k8s NotReady  master  2m   v1.8.4
```

Видите, только ведущий узел (мастер) указан как узел. И даже ведущий узел показан как неготовый. Вы поймете причину позже. Сейчас же вы займетесь настройкой своих двух узлов.

В.3 Настройка рабочих узлов с помощью kubeadm

При использовании `kubeadm` настройка рабочих узлов осуществляется даже проще, чем настройка ведущего узла. На самом деле, когда вы запускали команду `kubeadm init` для настройки ведущего узла, она уже сообщила вам, как настроить рабочие узлы (повторяется в следующем ниже листинге).

Листинг В.6. Последняя часть результата команды kubeadm init

```
You can now join any number of machines by running the following on each node
    as root:
kubeadm join --token eb3877.3585d0423978c549 192.168.64.138:6443
    --discovery-token-ca-cert-hash
    sha256:037d2c5505294af196048a17f184a79411c7b1eac48aaa0ad137075be3d7a847
```

На обоих рабочих узлах вам нужно лишь выполнить команду `kubeadm join` с указанным токеном и IP-адресом/портом ведущего узла. После этого потребуется меньше минуты, чтобы узлы зарегистрировались на ведущем узле. Вы можете получить подтверждение, что они зарегистрированы, снова выполнив команду `kubectl get node` на ведущем узле:

```
# kubectl get nodes
NAME        STATUS    ROLES   AGE  VERSION
master.k8s  NotReady  master  3m   v1.8.4
node1.k8s   NotReady  <none>  3s   v1.8.4
node2.k8s   NotReady  <none>  5s   v1.8.4
```

Хорошо, вы добились прогресса. Кластер Kubernetes теперь состоит из трех узлов, но ни один из них не готов. Давайте разбираться.

Давайте применим команду `kubectl describe` в следующем ниже листинге, чтобы увидеть больше информации. Где-то вверху вы увидите список `Conditions`, показывающий текущие условия на узле. Одно из них покажет следующую причину и сообщение.

Листинг В.7. Команда kubectl describe показывает, почему узел не готов

```
# kubectl describe node node1.k8s
...
KubeletNotReady    runtime network not ready: NetworkReady=false
                   reason:NetworkPluginNotReady message:docker:
                   network plugin is not ready: cni config uninitialized
```

Согласно ей, агент Kubelet готов не полностью, потому что плагин контейнерного сетевого взаимодействия (CNI) не готов, что вполне ожидаемо, поскольку вы еще не развернули плагин CNI. Сейчас вы его развернете.

В.3.1 Настройка контейнерной сети

Вы установите плагин контейнерного сетевого взаимодействия Weave Net, но, помимо него, также доступны несколько альтернатив. Они перечислены среди существующих надстроек Kubernetes по адресу http://kubernetes.io/docs/admin/addons/.

Развертывание плагина Weave Net (как и большинства других надстроек) выполняется так же просто, как следующая ниже команда:

```
$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl
    version | base64 | tr -d '\n')
```

Она приведет к развертыванию набора DaemonSet и нескольких ресурсов, связанных с безопасностью (см. главу 12 для объяснения кластерной роли ClusterRole и привязки кластерной роли ClusterRoleBinding, которые развертываются вместе с набором DaemonSet).

Как только контроллер DaemonSet создаст модули и они будут запущены на всех узлах, узлы должны быть готовы:

```
# k get node
NAME        STATUS  ROLES   AGE  VERSION
master.k8s  Ready   master  9m   v1.8.4
node1.k8s   Ready   <none>  5m   v1.8.4
node2.k8s   Ready   <none>  5m   v1.8.4
```

И на этом все. Теперь у вас есть полностью функционирующий трехузловой кластер Kubernetes с оверлейной сетью, предоставляемой плагином Weave Net. Все необходимые компоненты, за исключением самого агента Kubelet, выполняются как модули, управляемые агентом Kubelet, как показано в следующем ниже листинге.

Листинг В.8. Системные модули в пространстве имен kube-system после развертывания плагином Weave Net

```
# kubectl get po --all-namespaces
NAMESPACE    NAME                                READY  STATUS   AGE
kube-system   etcd-master.k8s                    1/1    Running  1h
kube-system   kube-apiserver-master.k8s          1/1    Running  1h
kube-system kube-controller-manager-master.k8s  1/1    Running  1h
kube-system kube-dns-3913472980-cn6kz           3/3    Running  1h
kube-system kube-proxy-hcqnx                    1/1    Running  24m
kube-system kube-proxy-jvdlr                    1/1    Running  24m
kube-system kube-proxy-qb709                    1/1    Running  1h
```

```
kube-system kube-scheduler-master.k8s          1/1    Running  1h
kube-system weave-net-58zbk                    2/2    Running  7m
kube-system weave-net-91kjd                    2/2    Running  7m
kube-system weave-net-vt279                    2/2    Running  7m
```

В.4 Использование кластера с локальной машины

До этого момента вы использовали `kubectl` на ведущем узле для связи с кластером. Возможно, вы захотите настроить экземпляр `kubectl` и на локальной машине.

Для этого с помощью следующей ниже команды необходимо скопировать файл /etc/kubernetes/admin.conf с ведущего узла на локальную машину:

```
$ scp root@192.168.64.138:/etc/kubernetes/admin.conf ~/.kube/config2
```

Замените IP-адрес на IP-адрес вашего ведущего узла. Затем укажите переменную среды `KUBECONFIG` на файл ~/.kube/config2 следующим образом:

```
$ export KUBECONFIG=~/.kube/config2
```

Агент Kubectl теперь будет использовать этот файл конфигурации. Для того чтобы вернуться к использованию предыдущей конфигурации, отмените настройку переменной среды.

Теперь все настроено на использование кластера с вашей локальной машины.

Приложение C

Использование других контейнерных сред выполнения

C.1 Замена Docker на rkt

Мы упоминали rkt (произносится рокит) несколько раз в этой книге. Как и Docker, эта контейнерная платформа запускает приложения в изолированных контейнерах, используя те же технологии Linux, что и Docker. Давайте посмотрим, чем rkt отличается от Docker и как ее попробовать в Minikube.

Первая интересная вещь относительно rkt заключается в том, что она напрямую поддерживает понятие модуля Pod (запуск нескольких связанных контейнеров), в отличие от Docker, который работает только с отдельными контейнерами. Контейнерная платформа rkt основана на открытых стандартах и с самого начала была сконструирована с учетом безопасности (например, образы подписываются, поэтому вы можете быть уверены, что они не подделаны). В отличие от платформы Docker, изначально имевшей клиент-серверную архитектуру, которая не очень хорошо справлялась с init-системами, такими как systemd, rkt – это инструмент CLI, который запускает ваш контейнер напрямую, вместо того чтобы поручать это делать демону. Приятная вещь относительно rkt заключается в том, что она может запускать существующие образы контейнеров в формате Docker, и поэтому, для того чтобы начать работу с rkt, вам не нужно переупаковывать свои приложения.

C.1.1 Настройка Kubernetes для использования rkt

Как вы помните из главы 11, агент Kubelet является единственным компонентом Kubernetes, который взаимодействует с контейнерной средой выполнения. Чтобы Kubernetes использовал платформу rkt вместо Docker, вам нужно сконфигурировать агента Kubelet, чтобы тот ее использовал, запустив его с параметром командной строки `--container-runtime=rkt`. Но имейте в виду, что поддержка rkt не такая зрелая, как поддержка Docker.

Пожалуйста, обратитесь к документации Kubernetes для получения дополнительной информации о том, как использовать rkt и что поддерживается или не поддерживается. Здесь мы рассмотрим краткий пример, просто чтобы начать работать с rkt.

С.1.2 Опробирование платформы rkt с Minikube

К счастью, для того чтобы начать работу с rkt на Kubernetes, от вас требуется лишь тот же самый исполняемый файл Minikube, который вы уже используете. Для применения rkt в качестве контейной среды выполнения в Minikube вам нужно лишь запустить Minikube с двумя параметрами:

```
$ minikube start --container-runtime=rkt --network-plugin=cni
```

> **ПРИМЕЧАНИЕ.** Вам может понадобиться выполнить команду `minikube delete`, которая удалит существующую виртуальную машину Minikube.

Параметр `--container-runtime=rkt` явным образом настраивает Kubelet на использование rkt в качестве контейнерной среды выполнения, тогда как параметр `--network-plugin=cni` заставляет его использовать контейнерный интерфейс сетевого взаимодействия в качестве сетевого плагина. Без этого параметра модули не будут работать, поэтому крайне важно его использовать.

Запуск модуля

После того как виртуальная машина Minikube поднята, вы можете взаимодействовать с Kubernetes в точности как раньше. Вы можете развернуть приложение `kubia` командой `run kubectl`, например:

```
$ kubectl run kubia --image=luksa/kubia --port 8080
deployment "kubia" created
```

Когда модуль запускается, вы можете увидеть, что он работает, посредством платформы rkt, проинспектировав его контейнеры с помощью команды `kubectl describe`, как показано в следующем ниже листинге.

Листинг С.1. Модуль, работающий с rkt

```
$ kubectl describe pods
Name:           kubia-3604679414-l1nn3
...
Status:         Running
IP:             10.1.0.2
Controllers:    ReplicaSet/kubia-3604679414
Containers:
  kubia:
```

```
    Container ID:    rkt://87a138ce-...-96e375852997:kubia   <─┐ Идентификаторы контейнера
    Image:           luksa/kubia                               │ и образа упоминают о rkt
    Image ID:        rkt://sha512-5bbc5c7df6148d30d74e0...   <─┘ вместо Docker
...
```

Вы также можете попробовать попасть в порт HTTP модуля, чтобы узнать, правильно ли он откликается на запросы HTTP. Это можно сделать, например, создав службу `NodePort` или применив команду `kubectl port-forward`.

Проверка запущенных контейнеров в виртуальной машине Minikube

Для того чтобы познакомиться с rkt поближе, попробуйте войти в виртуальную машину Minikube с помощью следующей ниже команды:

```
$ minikube ssh
```

Затем можно применить команду `rkt list` для просмотра работающих модулей и контейнеров, как показано в следующем ниже листинге.

Листинг С.2. Вывод списка работающих контейнеров с помощью команды rkt list

```
$ rkt list
UUID      APP                 IMAGE NAME                        STATE ...
4900e0a5  k8s-dashboard       gcr.io/google_containers/kun...   running ...
564a6234  nginx-ingr-ctrlr    gcr.io/google_containers/ngi...   running ...
5dcafffd  dflt-http-backend   gcr.io/google_containers/def...   running ...
707a306c  kube-addon-manager  gcr.io/google-containers/kub...   running ...
87a138ce  kubia               registry-1.docker.io/luksa/k...   running ...
d97f5c29  kubedns             gcr.io/google_containers/k8s...   running ...
          dnsmasq             gcr.io/google_containers/k8...
          sidecar             gcr.io/google_containers/k8...
```

Вы увидите работающий контейнер `kubia`, а также другие системные контейнеры (развернутые в модулях в пространстве имен `kube-system`). Обратите внимание, что в двух нижних контейнерах нет ничего в столбцах `UUID` или `STATE`. Это потому, что они принадлежат к тому же модулю, что и контейнер `kubedns`, указанный выше.

Платформа rkt печатает контейнеры, принадлежащие одному модулю, группируя их вместе. Каждый модуль (вместо каждого контейнера) имеет свой собственный UUID и состояние. Если вы попытались бы сделать это с платформой Docker, используемой в качестве контейнерной среды выполнения, то вы оцените, насколько с rkt проще увидеть все модули и их контейнеры. Вы заметите, что для каждого модуля не существует инфраструктурного контейнера (мы объяснили их в главе 11). Это вследствие нативной поддержки модулей в платформе rkt.

Вывод списка образов контейнеров

Если вы экспериментировали с командами Docker CLI, то вы быстро освоитесь с командами rkt. Запустите rkt без аргументов, и вы увидите все команды,

которые можете выполнить. Например, чтобы вывести список образов контейнеров, выполните команду из следующего ниже листинга.

Листинг C.3. Вывод списка образов с помощью команды rkt image list

```
$ rkt image list
ID           NAME                            SIZE    IMPORT TIME  LAST USED
sha512-a9c3  ...addon-manager:v6.4-beta.1    245MiB  24 min ago   24 min ago
sha512-a078  .../rkt/stage1-coreos:1.24.0    224MiB  24 min ago   24 min ago
sha512-5bbc  ...ker.io/luksa/kubia:latest    1.3GiB  23 min ago   23 min ago
sha512-3931  ...es-dashboard-amd64:v1.6.1    257MiB  22 min ago   22 min ago
sha512-2826  ...ainers/defaultbackend:1.0    15MiB   22 min ago   22 min ago
sha512-8b59  ...s-controller:0.9.0-beta.4    233MiB  22 min ago   22 min ago
sha512-7b59  ...dns-kube-dns-amd64:1.14.2    100MiB  21 min ago   21 min ago
sha512-39c6  ...nsmasq-nanny-amd64:1.14.2    86MiB   21 min ago   21 min ago
sha512-89fe  ...-dns-sidecar-amd64:1.14.2    85MiB   21 min ago   21 min ago
```

Все эти образы контейнеров имеют формат Docker. Вы также можете попробовать создать образы в формате образов OCI (OCI означает Open Container Initiative, Открытая контейнерная инициатива) с помощью инструмента `acbuild` (который можно получить по адресу https://github.com/containers/build) и запустить их с помощью rkt. Данная тема выходит за рамки этой книги, поэтому я позволю вам попробовать сделать это самостоятельно.

Информации, представленной в этом приложении до сих пор, должно быть достаточно, чтобы вы начали использовать rkt вместе с Kubernetes. Для получения дополнительной информации обратитесь к документации rkt по адресу https://coreos.com/rkt и документации Kubernetes на https://kubernetes.io/docs.

C.2 Использование других контейнерных сред выполнения посредством CRI

Поддержка в Kubernetes других контейнерных сред выполнения не останавливается на Docker и rkt. Обе эти среды выполнения изначально были интегрированы непосредственно в Kubernetes, но в Kubernetes версии 1.5 был представлен интерфейс контейнерной среды выполнения (Container Runtime Interface, CRI). CRI – это подключаемый API, позволяющий легко интегрировать другие контейнерные среды выполнения в Kubernetes. Теперь пользователи могут подключать к Kubernetes другие контейнерные среды выполнения без необходимости углубляться в код Kubernetes. От вас требуется только реализовать несколько интерфейсных методов.

С Kubernetes версии 1.6 и далее интерфейс CRI используется агентом Kubelet по умолчанию. Теперь и Docker, и rkt используются через CRI (не напрямую).

C.2.1 Знакомство с контейнерной средой выполнения CRI-O

Помимо Docker и rkt, новая реализация CRI под названием CRI-O позволяет Kubernetes напрямую запускать OCI-совместимые контейнеры и управлять ими, не требуя развертывания дополнительной контейнерной среды выполнения.

Вы можете попробовать CRI-O совместно с Minikube, запустив его с параметром `--container-runtime=crio`.

C.2.2 Запуск приложений на виртуальных машинах вместо контейнеров

Kubernetes – это система оркестровки контейнеров, верно? На протяжении всей книги мы исследовали множество функциональных возможностей, которые показывают, что она гораздо больше, чем система оркестровки, но суть в том, что при запуске приложения с помощью Kubernetes данное приложение всегда работает внутри контейнера, не так ли? Вы можете удивиться, что это больше не так.

Разрабатываются новые реализации CRI, позволяющие Kubernetes запускать приложения не в контейнерах, а на виртуальных машинах. Одна такая реализация, называемая Frakti, позволяет запускать обычные образы контейнеров на основе Docker непосредственно через гипервизор, что означает, что каждый контейнер запускает свое собственное ядро. Это позволяет гораздо лучше изолировать контейнеры по сравнению с тем, когда они используют одно и то же ядро.

И это еще не все. Еще одной реализацией CRI является Mirantis Virtlet, который позволяет вместо образов контейнеров запускать фактические образы виртуальных машин (в формате файла образов QCOW2, являющемся одним из форматов, используемых инструментом виртуальной машины QEMU). При применении Virtlet в качестве плагина CRI система Kubernetes запускает виртуальную машину для каждого модуля. Разве это не потрясающе?

Приложение D

Кластерная федерация

В разделе о высокой доступности в главе 11 мы рассмотрели, как Kubernetes может справляться с аварийными сбоями отдельных машин и даже аварийными сбоями целых серверных стоек или поддерживающей инфраструктуры. **Но что, если погаснет весь центр обработки данных?**

Для того чтобы гарантировать защиту от аварийных сбоев в дата-центре, приложения следует развертывать в нескольких дата-центрах или облачных зонах доступности. Когда один из этих центров обработки данных или зон доступности становится недоступным, клиентские запросы могут направляться приложениям, работающим в остальных работоспособных центрах обработки данных или зонах.

Хотя Kubernetes не требует, чтобы вы запускали плоскость управления и узлы в одном центре обработки данных, вы почти всегда захотите сделать это, чтобы сохранить сетевую задержку между ними на низком уровне и уменьшить возможность их разъединения друг от друга. Вместо единого кластера, расположенного в нескольких местах, лучше иметь отдельный кластер Kubernetes в любом месте. Мы рассмотрим этот подход в данном приложении.

D.1 Знакомство с кластерной федерацией Kubernetes

Экосистема Kubernetes позволяет объединять несколько кластеров в кластер кластеров посредством кластерной федерации. Она позволяет пользователям развертывать приложения и управлять ими в нескольких кластерах, работающих в разных точках мира, а также у разных облачных поставщиков в сочетании с локальными кластерами (гибридное облако). Целью кластерной федерации является не только обеспечение высокой доступности, но и объединение нескольких разнородных кластеров в один суперкластер, управляемый через единый интерфейс управления.

Например, объединяя локальный кластер с кластером, работающим в инфраструктуре поставщика облачных служб, можно запускать чувствительные к конфиденциальности компоненты локальной системы приложений на сервер организации, а нечувствительные части – в облаке. Еще одним примером является запуск приложения только в небольшом локальном

кластере, но когда вычислительные потребности приложения превышают емкость кластера, давая приложению возможность переходить в облачный кластер, который автоматически настраивается в инфраструктуре поставщика облачных служб.

D.2 Архитектура

Давайте рассмотрим, что такое кластерная федерация Kubernetes. Кластер кластеров можно сравнить с обычным кластером, в котором вместо узлов имеются полные кластеры. Так же, как кластер Kubernetes, который состоит из плоскости управления и нескольких рабочих узлов, федеративный кластер состоит из федеративной плоскости управления и нескольких кластеров Kubernetes. Подобно тому, как плоскость управления Kubernetes управляет приложениями в наборе рабочих узлов, федеративная плоскость управления делает то же самое, но в наборе кластеров вместо узлов.

Федеративная плоскость управления состоит из трех элементов:

- хранилища etcd для хранения федеративных объектов API;
- федеративного сервера API;
- федеративного менеджера контроллеров.

Это не сильно отличается от обычной плоскости управления Kubernetes. Хранилище etcd хранит федеративные объекты API, сервер API является конечной точкой REST, с которой взаимодействуют все другие компоненты, а федеративный менеджер контроллеров запускает различные федеративные контроллеры, выполняющие операции на основе объектов API, создаваемых через сервер API.

Пользователи обращаются к федеративному серверу API для создания федеративных объектов API (или федеративных ресурсов). Контроллеры федерации наблюдают за этими объектами и затем обращаются к серверам API базовых кластеров для создания обычных ресурсов Kubernetes. Архитектура федеративного кластера показана на рис. D.1.

D.3 Федеративные объекты API

Федеративный сервер API позволяет создавать федеративные варианты объектов, с которыми вы познакомились в этой книге.

D.3.1 Знакомство с федеративными версиями ресурсов Kubernetes

На момент написания этой книги поддерживаются следующие федеративные ресурсы:

- пространства имен Namespaces;
- словари конфигурации ConfigMap и секреты Secret;

- службы Service и входы Ingress;
- развертывания Deployment, наборы реплик ReplicaSet, задания Job и наборы реплик DaemonSet;
- автопреобразователи масштаба HorizontalPodAutoscaler.

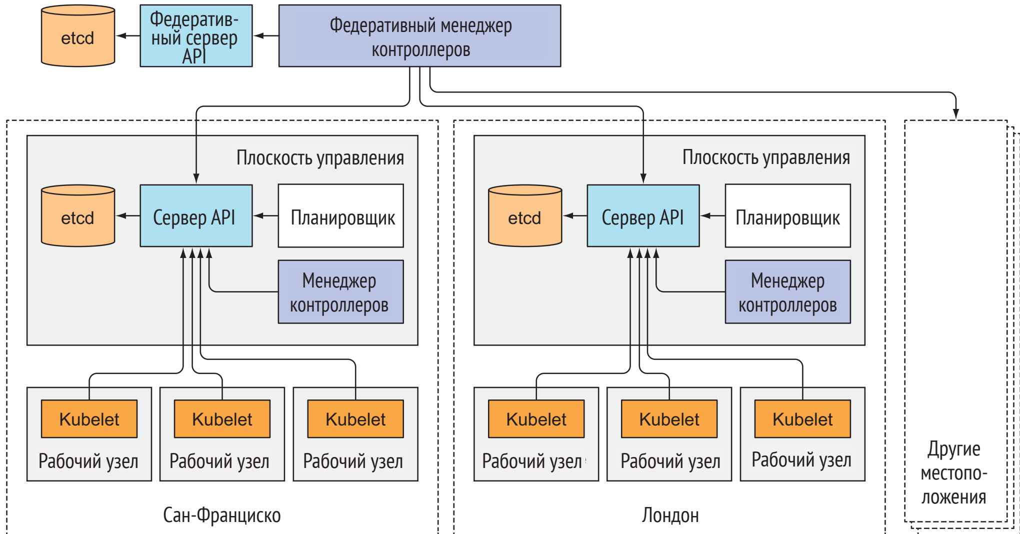


Рис. D.1. Кластерная федерация с кластерами в разных географических точках

> **ПРИМЕЧАНИЕ.** Относительно обновленного списка поддерживаемых федеративных ресурсов обратитесь к документации по кластерной федерации Kubernetes.

Помимо этих ресурсов, федеративный сервер API еще поддерживает объект Cluster, представляющий базовый кластер Kubernetes, так же, как объект Node представляет рабочий узел в обычном кластере Kubernetes. Для того чтобы визуализировать связь федеративных объектов с объектами, создаваемыми в базовых кластерах, изучите рис. D.2.

D.3.2 Что делают федеративные ресурсы

Для части федеративных объектов при создании объекта на федеративном сервере API контроллеры, работающие в федеративном менеджере контроллеров, будут создавать обычные внутрикластерные ресурсы во всех базовых кластерах Kubernetes и управлять ими до тех пор, пока федеративный объект не будет удален.

Для некоторых типов федеративных ресурсов ресурсы, созданные в базовых кластерах, являются точными репликами федеративного ресурса; для других это слегка измененные версии, в то время как некоторые федеративные ресурсы вообще ничего не создают в базовых кластерах. Реплики синхронизируются с исходными федеративными версиями. Но синхронизация

выполняется только в одном направлении – от федеративного сервера к базовым кластерам. При изменении ресурса в базовом кластере изменения не будут синхронизированы с федеративным сервером API.

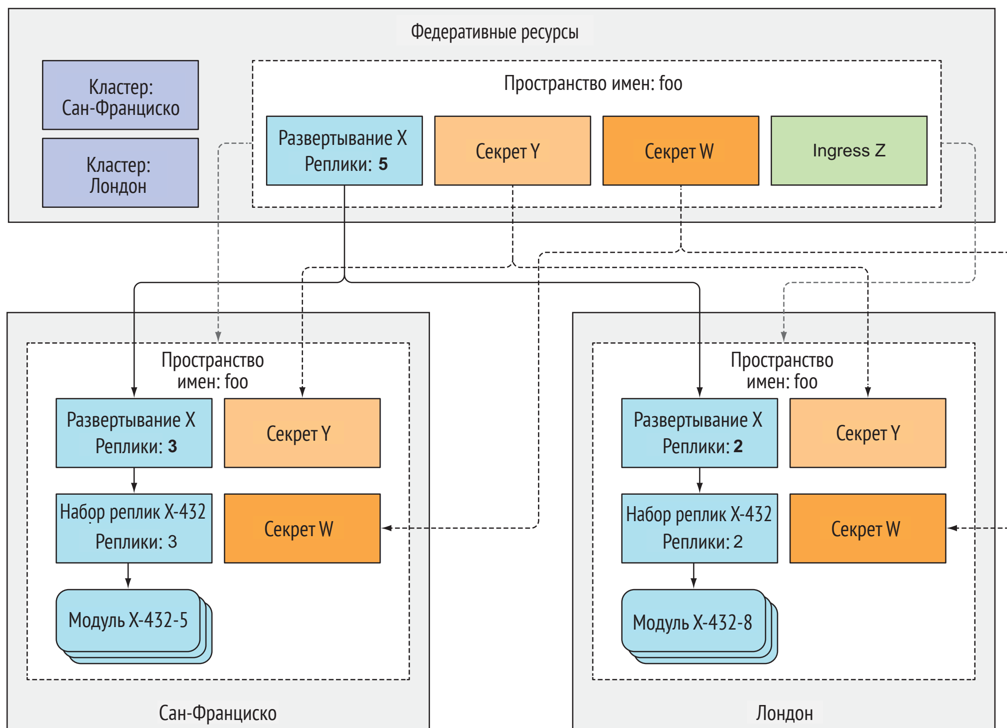


Рис. D.2. Взаимосвязь между федеративными ресурсами и обычными ресурсами в базовых кластерах

Например, при создании пространства имен на федеративном сервере API во всех базовых кластерах будет создано пространство имен с тем же именем. Если внутри этого пространства имен вы затем создадите федеративный словарь конфигурации ConfigMap, то ConfigMap с этим точным именем и содержимым будет создан во всех базовых кластерах, в том же пространстве имен. Это также относится к секретам, службам и наборам демонов.

Наборы реплик и развертывания отличаются. Они не копируются слепо в базовые кластеры, потому что это не то, что обычно хочет пользователь. В конце концов, если вы создаете развертывания с требуемым количеством реплик, равным 10, то вы, вероятно, не хотите, чтобы 10 реплик работали в каждом кластере. Всего нужно 10 реплик. По этой причине, когда вы указываете требуемое количество реплик в развертывании или наборе реплик, федеративные контроллеры создают базовые развертывания/наборы реплик так, чтобы сумма их требуемых количеств реплик равнялась требуемому количеству реплик, указанному в федеративных развертываниях или наборе реплик. По умолчанию реплики равномерно распределяются по кластерам, но это может быть переопределено.

> **ПРИМЕЧАНИЕ.** В настоящее время вам нужно подключаться к серверу API каждого кластера индивидуально для получения списка модулей, работающих в этом кластере. Вы не можете вывести список всех модулей кластеров через федеративный сервер API.

С другой стороны, федеративный ресурс Ingress не приводит к созданию каких-либо объектов Ingress в базовых кластерах. Из главы 5 вы, возможно, помните, что Ingress представляет собой единую точку входа в службу для внешних клиентов. По этой причине федеративный входной ресурс Ingress создает глобальную точку входа для служб во всех базовых кластерах на уровне нескольких кластеров.

> **ПРИМЕЧАНИЕ.** Что касается обычных входов Ingress, то для них требуется федеративный контроллер Ingress.

Настройка федеративных кластеров Kubernetes выходит за рамки этой книги, поэтому вы можете узнать по теме больше, обратившись к разделам кластерной федерации в руководстве пользователя и руководстве по администрированию в онлайновой документации Kubernetes по адресу http://kubernetes.io/docs/.

Предметный указатель

D

Docker и Kubernetes
- первые шаги
  - запуск первого приложения на Kubernetes, 81–94
  - настройка кластера Kubernetes, 73–79
  - создание, запуск и совместное использование образа контейнера, 61–72

K

Kubernetes
- знакомство
  - контейнерные технологии, 42–51
  - объяснение необходимости системы наподобие Kubernetes, 36–41
  - описание и преимущества, 51–59

А

автоматическое масштабирование модулей и узлов кластера
- вертикальное автомасштабирование модуля, 545–546
- горизонтальное автомасштабирование модуля, 529–444
- горизонтальное масштабирование узлов кластера, 546–550

архитектура
- планировщик, 393–396

В

внутренное устройство Kubernetes
- архитектура, 382–406
- взаимодействие контроллеров, 406–409
- запуск высокодоступных кластеров, 418–422
- интермодульное сетевое взаимодействие, 411–415
- реализация служб, 416–417
- что такое запущенный модуль, 411–412

Д

доступ к метаданным модуля и другим ресурсам из приложений
- обмен с сервером API Kubernetes, 296–313
- передача метаданных через нисходящий API, 287–295

З

защита сервера API Kubernetes
- аутентификация, 424–431
- защита кластера с помощью управления ролевым доступом, 432–455

защита узлов кластера и сети
- изоляция сети модулей, 485–489
- использование в модуле пространств имен хоста, 457–461
- конфигурирование контекста безопасности контейнера, 462–473
- ограничение использования функциональности, связанной с безопасностью в модулях, 474–484

К

контроллер репликации и другие контроллеры
- запуск модулей, выполняющих одну заканчиваемую задачу, 158–162
- запуск ровно одного модуля на каждом узле с помощью DaemonSet, 153–157
- знакомство с контроллерами репликации, 134–148
- использование набора реплик ReplicaSet вместо контроллера репликации, 149–152
- планирование выполнения заданий периодически или один раз в будущем, 163–165
- поддержание модулей в здоровом состоянии, 127–133

М

модули
- запуск контейнеров в Kubernetes
  - аннотирование модулей, 116–117
    - добавление и изменение аннотаций, 117–118
    - поиск аннотаций объекта, 117
  - знакомство с модулями, 94–99
    - зачем нужны модули, 95–96
    - общее представление о модулях, 96–97

правильная организация контейнеров между модулями, 98–99
использование меток и селекторов для ограничения планирования модулей, 114–116
использование меток для классификации рабочих узлов, 115
планирование размещения на один конкретный узел, 116
приписывание модулей к определенным узлам, 115
использование пространств имен для группирования ресурсов, 118–121
изоляция, обеспечиваемая пространствами имен, 121
необходимость пространств имен, 118–119
обнаружение других пространств имен и их модулей, 119–120
создание пространства имен, 120–121
управление объектами в других пространствах имен, 121
организация модулей с помощью меток, 108–111
знакомство с метками, 108–109
изменение меток существующих модулей, 111
указание меток при создании модуля, 110–111
остановка и удаление модулей, 122–124
удаление (почти) всех ресурсов в пространстве имен, 124
удаление всех модулей в пространстве имен при сохранении пространства имен, 123
удаление модулей путем удаления всего пространства имен, 123
удаление модулей с помощью селекторов меток, 122
удаление модуля по имени, 122
перечисление подмножеств модулей посредством селекторов меток, 112–113
вывод списка модулей с помощью селектора меток, 112
использование нескольких условий в селекторе меток, 113
создание модулей из дескрипторов YAML или JSON, 100–107
использование команды kubectl create для создания модуля, 105–106
исследование дескриптора YAML существующего модуля, 101–102
отправка запросов в модуль, 107
просмотр журналов приложений, 106
создание простого дескриптора YAML для модуля, 103–104

П

приложения
использование kubectl с несколькими кластерами, 639–644
использование других контейнерных сред выполнения, 659–663
кластерная федерация, 664–666
настройка многоузлового кластера с помощью kubeadm, 645–658
продвинутое назначение модулей узлам
использование ограничений и допусков для отделения модулей от определенных узлов, 552–557
использование сходства узлов для привлечения модулей к определенным узлам, 558–564
назначение модулей на удалении друг от друга с помощью антисходства модулей, 571–572

Р

развертывания
декларативное обновление приложений
выполнение автоматического плавного обновления с помощью контроллера репликации, 319–326
использование развертываний для декларативного обновления приложений, 327–347
обновление приложений, работающих в модулях, 316–318
расширение системы, 610–638
Deis Workflow и Helm, 635–637
автоматизация пользовательских ресурсов с помощью пользовательских контроллеров, 615–619
брокеры служб и API OpenServiceBroker, 625–626
валидация пользовательских объектов, 620
каталог служб, 631
контейнерная платформа Red Hat OpenShift, 631–634

определение своих собственных объектов API, 610–622
определения CustomResourceDefinition, 611–614
отвязывание и дерезервирование, 630
платформы поверх Kubernetes, 631–637
предоставление пользователького сервера API, 621–622
расширение с помощью каталога служб (Kubernetes Service Catalog), 623–630
резервирование экземпляра службы ServiceInstance, 627–628
сервер API каталога служб и менеджер контроллеров, 624–625
рекомендации по разработке приложений
жизненный цикл модуля, 576–590
обеспечение правильной обработки всех клиентских запросов, 591–596
упрощение запуска приложений и управления ими, 597–602
ресурсы StatefulSet
использование набора StatefulSet, 360–370
как наборы StatefulSet справляются с аварийными сбоями узлов, 377–380
набор модулей с внутренним состоянием, 349–352
обнаружение соседей в наборе StatefulSet, 371–376
репликация модулей с внутренним состоянием, 349–352

С

словари конфигурации и секреты
настройка приложений
использование секретов для передачи чувствительных данных в контейнеры, 274–284
конфигурирование контейнерных приложений, 248–253
настройка переменных среды для контейнера, 254–256
отсоединение конфигурации с помощью словаря конфигурации ConfigMap, 257–273
передача в контейнеры аргументов командной строки, 250–253
службы
обеспечение клиентам возможности обнаруживать модули и обмениваться с ними информацией
знакомство со службами, 168–179
использование служб без обозначенной точки входа, 205–208
обеспечение доступа к службам извне через ресурс Ingress, 192–199
подключение к службам, находящимся за пределами кластера, 179–184
предоставление внешним клиентам доступа к службам, 180–182
сигналы о готовности модуля к приему подключений, 200–204
устранение неполадок в службах, 208

Т

тома
динамическое резервирование томов PersistentVolume, 240–245
доступ к файлам в файловой системе рабочего узла, 222–223
знакомство с томами, 212–214
использование постоянного хранилища, 224–229
использование томов для обмена данными между контейнерами, 215–221
отделение модулей от базовой технологии хранения, 230–239

У

управление вычислительными ресурсами модулей
запрос на ресурсы для контейнеров модуля, 491–499
классы QoS модулей, 505–509
лимитирование общего объема ресурсов, 514–519
лимитирование ресурсов, доступных контейнеру, 500–504
мониторинг потребления ресурсов модуля, 520–525
установка стандартных запросов и лимитов для модулей, 510–513

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: тел**. +7(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Марко Лукша

Kubernetes в действии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Научные редакторы *Потапов Е. А., Хасанов Т. М.*
Перевод с английского *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 $^{1}/_{16}$. Печать цифровая.
Усл. печ. л. 54,6. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Kubernetes в действии

Kubernetes по-гречески означает “рулевой”. Это ваш проводник по неизведанным водам. Система контейнерной оркестровки Kubernetes безопасно управляет структурой распределенного приложения и последовательностью его выполнения, с максимальной эффективностью организуя контейнеры и службы. Kubernetes служит в качестве операционной системы для ваших кластеров, устраняя необходимость учитывать лежащую в основе сетевую и серверную инфраструктуру в ваших проектах.

Эта книга учит использовать Kubernetes для развертывания распределенных контейнеризированных приложений. Перед тем, как собрать свой первый кластер Kubernetes, вы начнете с обзора систем Docker и Kubernetes. Вы будете постепенно расширять свое начальное приложение, добавляя новые функциональные возможности и углубляя свои знания архитектуры и принципа работы Kubernetes. Также вы изучите такие важные темы, такие как мониторинг, настройка и масштабирование.

Краткое содержание:

- внутреннее устройство Kubernetes;
- развертывание контейнеров в кластере;
- обеспечение защиты кластеров;
- обновление приложений с нулевым временем простоя.

«В практическом стиле автор учит, как управлять полным жизненным циклом любого распределенного и масштабируемого приложения».

— Антонио Магнаги, System1

«Лучшее, что можно представить, — это примеры из реального мира. Они не просто применяют концепции, они подвергаются дорожному испытанию».

— Паоло Антинори, Red Hat

«Углубленное обсуждение Kubernetes и родственных технологий. Крайне необходимая вещь!»

— Эл Кринкер, USPTO

«Полный путь к становлению профессиональным Кубернотом. Существеннейшая книга».

— Чаба Сари, Chimera Entertainment

Марко Лукша (Marko Lukša)— инженер Red Hat, работающий на Kubernetes и OpenShift.

Написано для разработчиков программного обеспечения промежуточного уровня с небольшими или отсутствующими познаниями в Docker или других системах оркестровки контейнеров.

Интернет-магазин: www.dmkpress.com
Книга – почтой: orders@alians-kniga.ru
Оптовая продажа: “Альянс-книга”
тел.(499)782-3889. books@alians-kniga.ru

www.дмк.рф

ISBN 978-5-97060-657-5